AD-A008 877

DATACOMPUTER PROJECT

Computer Corporation of America

Prepared for:

Defense Supply Service
Advanced Research Projects Agency

31 December 1974

AD-A008877

Computer Corporation of America
575 Technology Square
Cambridge, Massachusetts 02139

DATACOMPUTER PROJECT
SEMI-ANNUAL TECHNICAL REPORT

July 1, 1974 to December 31, 1974

This research was supported by the Defense Advanced Research Projects Agency of the Department of Defense and was monitored by the U.S. Army Research Office, Defense Supply Service--Washington under Contract No. MDA903-74-C-0225. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

(110)

Abstract

The Datacomputer system is being designed as a large-scale data storage utility to be accessed from remote computers on the Arpanet and, potentially, on other networks. The development is phased, with each successive release of the system offering increased capabilities to users. During the present reporting period, the third major release of the system became operational. This release, while still primitive in many respects, is providing service for a wide range of applications.

Table of Contents

1. Overview

1.1 Review of Basic Concepts

The goal of the project continues to be the development of a shared, large-scale data storage utility, to serve the needs of the Arpanet community.

The system under development will make it possible to store within the network such files as the ETAC Weather File or the NMRO Seismic Data File, which are measured in hundreds of billions of bits, and to make arbitrarily selected parts of these files available within seconds to sites requesting the information. The system is also intended to be used as a centralized facility for archiving data, for sharing data among the various network hosts, and for providing inexpensive on-line storage to sites which need to supplement their local capability.

Logically, the system can be viewed as a closed box which is shared by multiple external processors, and which is accessed in a standard notation, "datalanguage" (see Fig. 1). The processors can request the system to store information, change information already stored in the system, and retrieve stored information. To cause the Datacomputer to take action, the external processor sends a "request" expressed in datalanguage to the Datacomputer, which then performs the desired data operations.

From the user's point of view the Datacomputer is a remotely-located utility, accessed by telecommunications. It would be impractical to use such a utility if, whenever the user wanted to access or change any portion of his file, the entire file had to be transmitted to him. Accordingly, data management functions (information retrieval, file maintenance, backup, access security, creation of direct and inverse files, maintenance of file directories, etc.) are performed by the Datacomputer

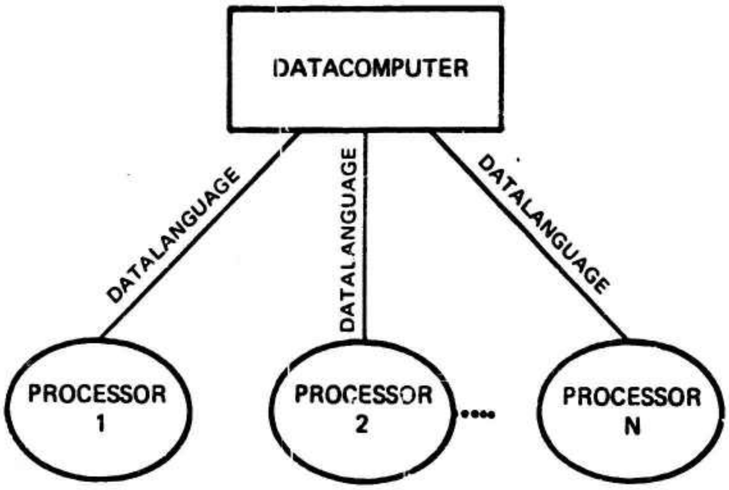

Figure 1. Logical View of Datacomputer

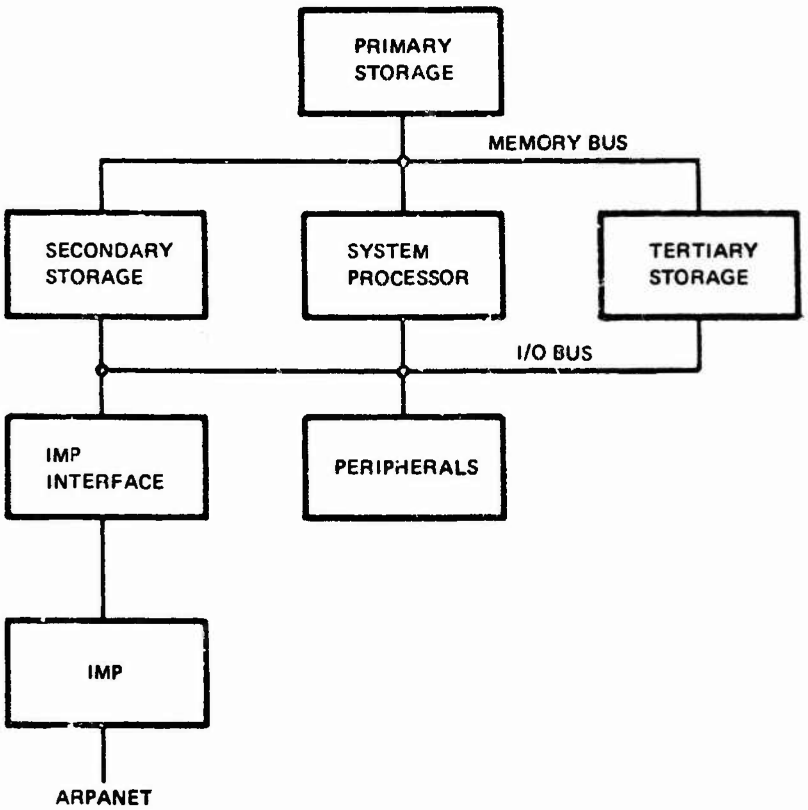

Figure 2. Hardware Overview of System

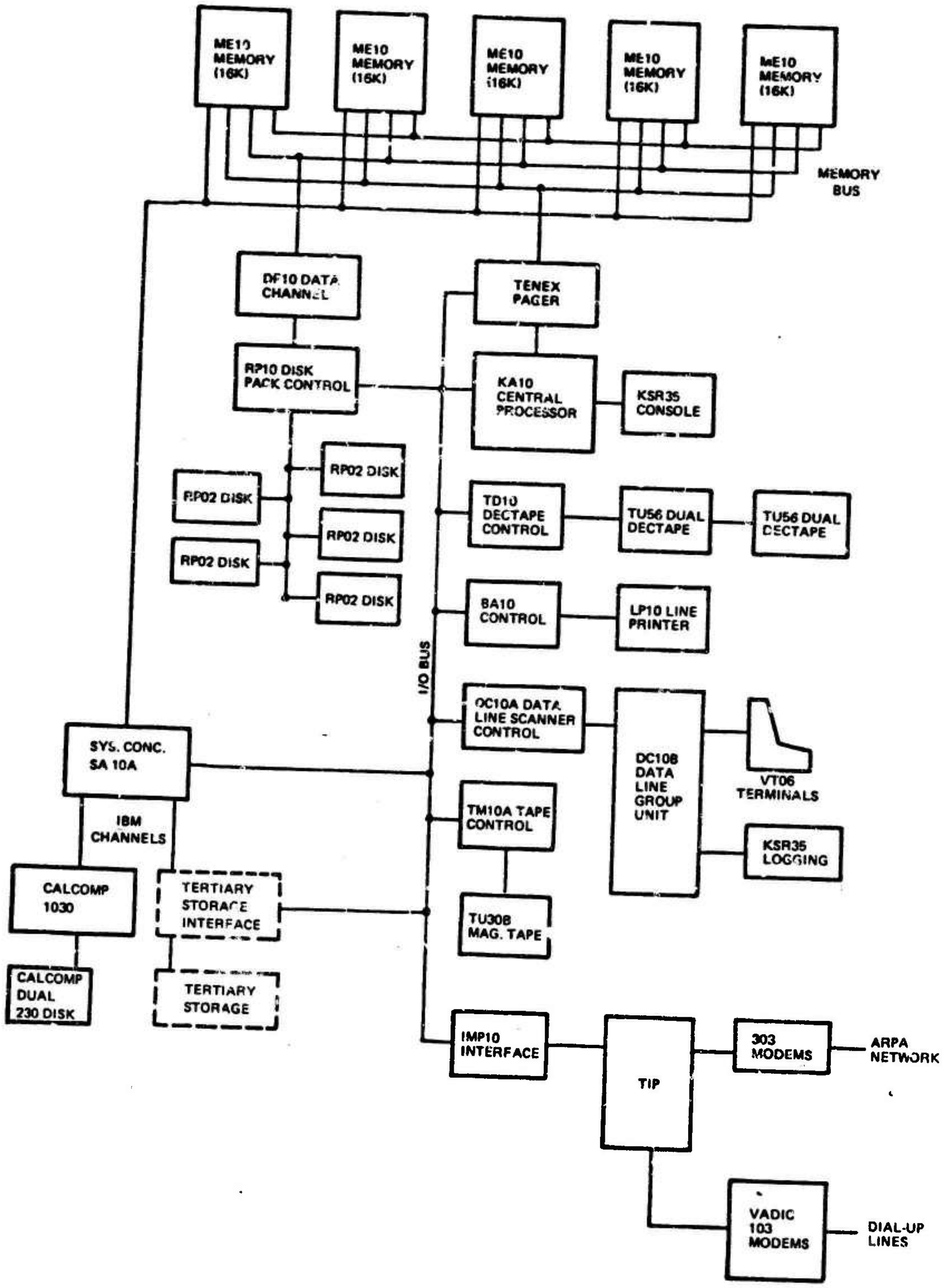

Figure 3. Hardware Block Diagram - CCA Installation
(Equipment in dashed outline is planned for 1975)

system itself. The user sends a "request", which causes the proper functions to be executed at the Datacomputer without requiring entire files to be shipped back and forth.

The hardware of the system is shown in overview in Fig. 2 and in greater detail in Fig. 3.

The program for the system processor handles the interactions with the network hosts and is designed to control up to three levels of storage: primary (core), secondary (disk), and tertiary (mass store). Currently, the CCA facility is operating with primary and secondary storage only, with the addition of tertiary storage planned for 1975. Installation of a tertiary storage module will leave datalanguage unchanged, and will therefore be imperceptible to users of the system (except insofar as it affects performance and the total storage capacity available for data).

In addition to using the dedicated equipment at CCA, it is planned that Datacomputer service will also make use of hardware resources located at NASA/Ames, using CCA software. The two sites will provide mutual backup for one another, thereby guarding against accidental loss of data and providing for satisfactory uptime of the overall service.

1.2 Status of Project

During this reporting period, Version 0/11 of the Datacomputer system was completed. This is the third major version of the system to offer Datacomputer services on the Arpanet.

New facilities in Version 0/11 include updating of fixed-length containers, inversions on variable-length containers and retrieval by index number. (See Chapter 2 and Appendix for details.)

The next version, 1/0, will be the first complete, though minimal, database management system. Its main features are full updating capabilities and concurrent updating and reading of a file. Version 1/0 will be completed in June 1975.

The project continues to interact with actual and potential Datacomputer users. A user's meeting was held to identify user needs so that they may be considered in setting implementation priorities. Several new applications are beginning, and major presentations have been made to potential users.

A paper giving an overview of the Datacomputer system was written. In addition to serving as a chapter of the Arpanet book, it will be presented at the National Computer Conference in May 1975.

2. Software Implementation

During this period, version 0/11 replaced version 0/10 as the Datacomputer system offering service on the Arpanet. The new features of 0/11 are summarized in this section. (See Appendix, "Datacomputer Version 0/11 User Manual" for details.) Specification and implementation of version 1/0 were begun.

2.1 Request Handler

The following enhancements were released in version 0/11:

1. Simple Updating

The user may specify value replacement for fixed length, uninverted fields. The update request takes a master file, and either a list of transactions or a constant. Since the Datacomputer is making a sequential pass of port and file, the information appearing in the transaction port must occur in the same order as it appeared in the master file; that is, outer and inner port list members must be in the same order as those of the file. Lists which contain no information different from the master file (which are not being changed by the update) may be omitted.

2. Virtual Index

A virtual container is one whose value is not stored as data, but can be derived. The kind of virtual container implemented in 0/11 is the virtual index container: the position, by number starting at 1, of a list-member within the list. For the virtual index container to be used, it must be included in the creating description.

3. Integer Type

Two's complement integers have been added as a data type in 0/11. Sizes may be specified up to 36 bits. Conversion (to and from ASCII strings) and comparison have been implemented for integers.

4. CAT

The Container Address Table (CAT) is a new internal structure in version 0/11 which will speed retrieval of variable length data. The CAT is a table of pointers to the start of each variable-length list member. The CAT is automatically formed for those variable length lists which contain at least one inverted string. The user can specify a CAT for lists which do not contain an inverted string. The CAT will be used for retrievals based on the virtual index container for files.

Definition and design of release 1/0, currently scheduled for June 1975, was begun in this period. Removal of the zero prefix implies that 1/0 will be the first "service offering" of the Datacomputer. As such, we are making a special commitment to "clean up" and "flush out" the system. Removal of restrictions imposed in previous releases will be a major goal of 1/0. The major features of this release will be:

1. Full Updating Capability

The user will be able to change the value of any elementary container, whether fixed or variable length, or inverted. Append and delete facilities will be available for lists at any level. Some ability to add list members in an ordered list will be implemented, but the ordering will be maintained by the user.

2. Intermediate Language

The method of compiling the high level datalanguage has been changed. A new formalism, intermediate language, has been

defined. The compilation process now translates datalanguage into intermediate language, and then compiles intermediate language into tuples (the system's most primitive operations). This new method allows more advanced compile time optimization, and makes the handling of datalanguage side effects more manageable. These side effects were responsible for many of the restrictions in previous releases of the system. Intermediate language also provides an internal base language so that addition of high level operators to the user datalanguage is greatly simplified.

3. Improved Directory List Features

The system will provide more information to the user about stored data, including times created, read, modified and size information on both base and inversion areas. Status information on open files will be provided. All this information will be available over the datalanguage (or control) path as well as via system-maintained descriptions over data paths. This allows user program control over format and content of list output.

2.2 Services

The Services subsystem of the Datacomputer has made significant progress both in overall design and in code implementation.

Design progress has been made in the following areas:

- SDAX (Special Disk Area Index) is the scheme for buffering on 3330 disk file data pages which will normally reside on tertiary memory. The data pages for files currently active will be buffered in the Special Disk Area (SDA) until the file is no longer in use. This scheme is expected to yield substantial efficiencies in internal data handling. The design is complete and detailed. Implementation is expected to take place over the next six to eight months.

- The RESERVE function has been designed and pre-implementation work has been done for the purpose of accomodating this design. This function will permit users to gain exclusive access to specified components of the file directory such as individual pathnames and privilege typle chains.

Extremely high priority is being placed upon verifiable integrity of the database. To this end, every underived item of directory and volume-descriptor information will be multiply redundant. The validation routines will have the capability of cross-checking redundant items, isolating faulty data fields, and, under programmer control, correcting such errors. Portions of this design are now complete, and are currently being implemented.

Implementation includes the following significant work:

- 32-bit mode. Since the Unicon is a 32-bit wide device, and TENEX files/memory utilizes 36-bit words, files stored on this device must not use the low 4 bits of each word. This capability has required the following changes to SV: (a) Added a new Datacomputer device, the "Unicon file", which is just like a TENEX file except that its VTOC is only 32 bits wide; (b) Required changing VTOC bit manipulation subroutines to accomodate 32-bit sized words; (c) Also required formulating and coding a new internal representation for date/times, since the TENEX standard requires 36 bits; (d) Since the width of a file is no longer implicitly 36, changes were made to the space allocation and deletion routines to pass the width as a parameter whenever manipulating the free chunk bp.

- A "slosh" program has been implemented for the purpose of transferring data from one Datacomputer to another. This greatly facilitates transfer of file directories and data between different (and sometimes incompatible) releases of Datacomputer software. Also, it has the potential to be used to transfer files between physically discrete Datacomputers.

- Directory system validation routines. This collection of routines may be run at any time in order to validate, and to perform extensive internal verification of, directory system information. Work is still being performed in this area.

3. Network Services

During the present reporting period, emphasis has been placed on getting operational experience with a range of Datacomputer applications.

One of the largest databases to be stored on the Datacomputer is the seismic data. The amount of data to be stored necessitates that the data be handled as efficiently as possible if the application is to be feasible. CCA has worked closely with the other organizations involved to identify the data storage and retrieval requirements for this application and to design the file formats. We have obtained several sample seismic files for experimental purposes. One of these files, containing data from the International Seismic Month, was used in a demonstration at an international conference. The user program SMART* was modified to access this data.

CCA took part in the ARPA intelligence show. A demonstration of the Datacomputer and a datalanguage tutorial were given. Accounts were set up to allow experimental access for potential DOD users.

As a part of the MIT-DMS message archival project, the Datacomputer will provide archival storage for network mail. Several meetings have taken place between CCA and DMS in order to design the application.

---

* SMART generates datalanguage for users at terminals, thus providing them with convenient access to the Datacomputer.

Appendix

Datacomputer Version 0/11 User Manual

Table of Contents

Appendices

Chapter 1: Introduction to the Datacomputer

Introduction

The datacomputer is a shared large-scale data utility system designed to serve the computers on the ARPA network. It may be thought of as a "black box" that performs data storage and retrieval functions in response to commands phrased in a standard notation, called datalanguage. The datacomputer in its full implementation will provide an on-line storage capacity of one trillion bits and an extensive set of services to user programs.(1)

This document describes Version 0/11, the currently running version of the datacomputer system, and includes information about how a user program can access the system, transmit datalanguage, process the datacomputer's responses, and transmit and receive data over the network.

Version 0/11 is a preliminary version. While it is sufficiently powerful for many types of applications, it still lacks some desirable features, and contains some undesirable implementation restrictions. The next version, 1/0, will offer a more complete subset of datalanguage. In particular, it is planned that full updating facilities will be available and that restrictions of the type described in Appendix E will be removed. Versions beyond 1/0 will progressively enlarge the range of services.

---

(1) See Datacomputer Project Working Paper No. 8, _Further Datalanguage Design Concepts_, December 1973.

Chapter 2: Containers

Containers

The <u>container</u> is a basic concept in datalanguage. A container is an imaginary box which, like a FORTRAN variable, may contain data; a container may also enclose other containers. For example, some information about people could be represented as:

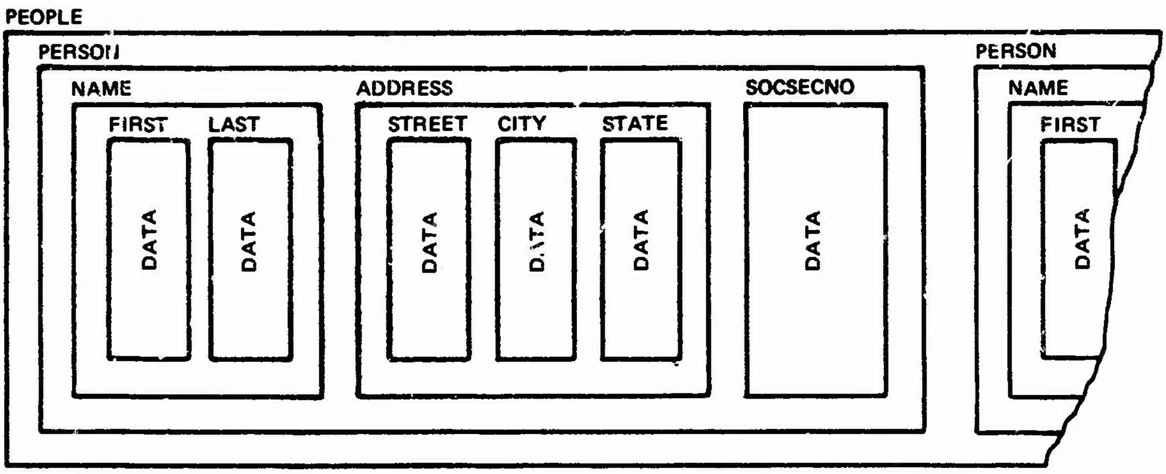

Figure 2-1. A container structure

Here PEOPLE, PERSON, NAME, and ADDRESS are containers enclosing other containers; FIRST, LAST, STREET, CITY, STATE, and SOCSECNO are containers that enclose only data.

The description of a container has several parts. It includes the container's ident, type, and size, and perhaps some additional attributes. The container's <u>ident</u>, or simple name, is a string of 100 or fewer letters, digits or the special character %, by which datalanguage requests refer to the container. The first character of an ident must be a letter or the character %. Certain reserved words may not be used as container idents; these are listed in Appendix B of this document.

Some sample idents are:

AVERYLONGIDENTABCDEFGHIJKLMNOPQRSTUVWXYZ
PEOPLE
WEATHEROSTATIONS
$CCA

Containers are of four _types_, depending on their contents.

A container that is a _LIST_ contains some number of other containers. The LIST-members may be containers of any data type, but they must all have the same description. PEOPLE (above) is an example of a LIST.

A container that is a _STRUCT_, or _STRUCTURE_, contains some number of other containers, which need not have identical descriptions.(1) The descriptions of all the containers that are enclosed by the STRUCT form part of the description of the STRUCT itself; and on every occurrence of the STRUCT every one of its sub-containers must appear in the same order. ADDRESS is an example of a STRUCT.

An elementary data container of type _BYTE_ or _INTEGER_ (_INT_) contains one byte of data. An elementary data container of type a _STR_ or _STRING_ contains a string of bytes.(2) The user can specify the byte size of BYTEs, INTs, and STRs and can indicate that STRs are 7- or 8-bit ASCII or uninterpreted. BYTEs are always uninterpreted, INTs are 2's complement. (See below).

A LIST or STR has a _size_ associated with it. The size may be fixed or variable. The size of a STR is the number of bytes in it, while the size of a LIST is the number of elements in the LIST.

Outermost Containers

A container that is not contained by any other container is called an _outermost container_; outermost containers are different in several respects from other containers.

An outermost container in datalanguage has a _function_, which is either FILE, PORT, or TEMPORARY PORT (which may be abbreviated TEMP PORT). A FILE contains data kept in the datacomputer. When a FILE is created (see below),

---

(1) STRUCT and STRUCTURE are synonyms in datalanguage. Hereafter, STRUCT will normally be used.

(2) INT and INTEGER, STR and STRING are synonyms in datalanguage. Hereafter, STR and INT will normally be used.

datacomputer space is allocated for it. A PORT describes data that is transmitted to or from the datacomputer. A TEMP PORT is a PORT whose description is not permanently stored, unlike the descriptions of other containers. The description of TEMP PORTs are deleted at the end of the session in which they were created.

The Directory

The ident of an outermost container, whether it is a FILE or a PORT, is unlike other idents, in that it is entered in the datacomputer's directory. The directory is conceptually a tree; the entries in it are called nodes. A node may have one or more subordinate nodes, unless it represents a container, in which case it cannot. A portion of a hypothetical directory is diagrammed below; it may be read as indicating that the nodes F and G are subordinate to DATA, which in turn is subordinate to CCA. Only the bottom-most nodes in this tree, F and G, may represent containers, and they represent outermost containers.

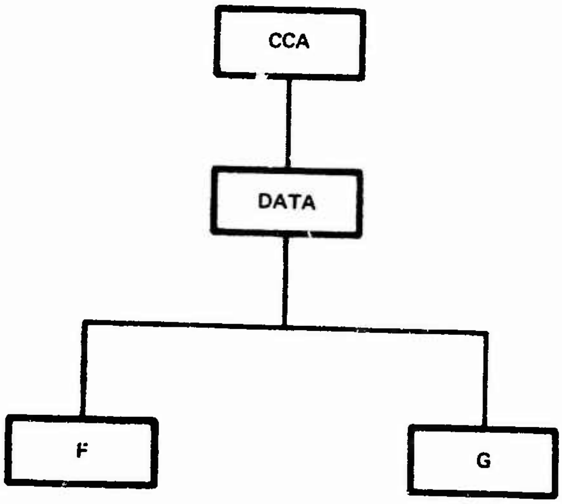

Figure 2-2. A portion of the directory.

Only a bottom-most node of the directory may be a container ident; only an outermost container has its ident entered in the directory.

Normally, the first thing a user does after connecting to the datacomputer is LOGIN to a directory node. For most purposes, he only sees his login node and the part of the directory that is subordinate to his login node. (The LOGIN request is discussed in detail in Chapter 4.)

Pathnames

Pathnames are used to reference nodes in the directory tree by describing a path through it. They have the general hierarchical form

    NODE1.NODE2...NODEn

where NODE2 is a node directly subordinate to NODE1.

There are several varieties of pathnames. The two classes of directory objects referenced by pathnames are closed nodes (including all nodes that are not outermost containers and therefore cannot be open, and all outermost containers that are not OPEN) and OPEN outermost containers. There are three areas in which names can be found: the TOP, LOGIN and OPEN contexts. Thus there are six possible pathname types, only five of which are reasonable. (A closed node in the OPEN context isn't.)

Closed nodes can be referenced either by a complete pathname (started with the reserved word %TOP), which causes the name search to be anchored at the top of the directory tree, or a LOGIN pathname, which anchors the search at the current LOGIN node. Either pathname may contain passwords. (Passwords are discussed in chapter 4.)

OPEN nodes may be referenced by a simple complete pathname or a simple LOGIN pathname, neither of which can contain passwords, or by an OPEN node simple name. An OPEN node simple name is the name of the outermost container.

Creating Nodes

A node in the directory is created with a CREATE request. Such a request has the form

    CREATE <pathname> ;

Only one node may be created by a single CREATE request, and a higher-level node must always be created before one subordinate to it. The reserved words listed in Appendix B may not be used as directory node names.

As an example, let us create the outermost container F, a LIST of 4-character strings; the container's ident will be entered in the directory as indicated in Figure 2-2. We assume that nothing is presently in the directory, so we must start by creating the topmost node.

```
CREATE CCA;
CREATE CCA.DATA;
```

```
CREATE CCA.DATA.F FILE LIST
                 XYZ STR (4);
```

Now that CCA and CCA.DATA have been created, we could create CCA.DATA.G with only one CREATE request; i.e.

```
CREATE CCA.DATA.G PORT LIST  etc.
```

Creating Containers

Outermost containers are created by a more complicated form of the CREATE request. The CREATE statement must tell the datacomputer all about the container. An outermost container and all its subcontainers must be created at once, with one CREATE request.

The CREATE request causes the description to be stored. It also causes space to be allocated if the container is a FILE.

The full BNF in Appendix A indicates succinctly the precise syntax of the CREATE statement. It is worth looking at a few examples before looking at all the details of descriptions. One example, a LIST of STRings:

```
CREATE ALPHA FILE LIST SUBCONTAINEDSTRING STR (44);
```

Here the size of the outermost LIST is omitted, so the datacomputer will calculate a default size.

A LIST of STRUCTs, each of which contains three strings:

```
CREATE BALLTEAM FILE LIST (25)
        PLAYER STRUCT
          NAME STR (20)
          POSITION STR (2)
          UNIFORM%NUMBER STR (2)
        END;
```

The datacomputer will allocate enough space for the file BALLTEAM to hold 25 copies of the STRUCT named PLAYER. NAME, POSITION and UNIFORM%NUMBER are 7-bit ASCII STRs. note that END is required to terminate the description of the STRUCT.

The example diagrammed on page 4:

```
CREATE PEOPLE FILE LIST
  PERSON STRUCT
   NAME STRUCT
     FIRST STR (15)
```

```
    LAST STR (15)
  END
  ADDRESS STRUCT
    STREET STR (15)
    CITY STR (15)
    STATE STR (15)
  END
  SOCSECNO STR (10)
END;
```

The elementary data types are BYTE, INT and STR. Containers of these types contain data, not other containers. INT is a 2's complement integer, while BYTE is uninterpreted.

STRings and LISTs must have a size. For LISTs the size is the number of LIST members (e.g., the number of PERSONs in PEOPLE above.) The three forms for indicating the size are:

```
(n) -- a fixed size of n
(m,n) -- a minimum size of m and a maximum of n
(,n) -- a minimum dimension of 0 and a maximum of n
```

where m and n are positive integers.

<u>Terminator Options:</u> For an outermost LIST or STRing, no size need be specified. For a FILE, the default minimum is 0, and the default maximum is based on what will fit in the default space allocation. For a PORT, the default minimum is also zero, but the default maximum is effectively infinite.

The datacomputer needs a way to find the end of the data in variable-sized LISTs and STRings. The three options are a preceding count, a trailing delimiter, and punctuation (i.e., a device-dependent marker). A one-byte preceding count is indicated with the keyword parameter:

```
,C=1
```

Version 0/11 cannot handle counts larger than one byte. Thus, if there is a count, then the maximum dimension must be small enough to fit into a one-byte count. (Byte size is is discussed further below.) The value of the count does not include the count byte itself.

The syntax to indicate that there is a one-byte delimiter is:

```
,D=n
```

or

```
,D='a'
```

where n is a decimal number and 'a' is any ASCII number, letter or special character.

The datacomputer considers punctuation for ASCII PORTs to be different from delimiters. Punctuation over the network is a special character (specifically EOR, EOB, or EOF) inserted in the data but not considered part of the data. This is indicated by:

```
,P=EOR     (carriage return, line feed)
,P=EOB     (Control-L)
```

and

```
,P=EOF     (Control-Z)
```

A fixed-size container (including a STRUCT) in a PORT, may have a P, D, or C parameter, but no container (fixed or variable) may have more than one of these. PORT's and FILEs may not have an outer level D or C parameter.

A FILE may be punctuated with an EOF, but the datacomputer ignores this punctuation. No subcontainers of a FILE may be punctuated. Variable length subcontainers of a FILE must have either a C (count) or D (delimited) parameter, fixed length may have.

If a variable-sized PORT does not have an outer level P parameter specified, then it defaults to P=EOF. Variable-sized subcontainers of a PORT must have a C or D parameter, or be punctuated. A subcontainer of a PORT may have a C parameter only if the PORT is a secondary PORT (see Chapter 6).

Punctuation is hierarchical. A container that is punctuated with EOR cannot contain one that is punctuated with EOR, EOB or EOF. A container that is punctuated with EOB cannot contain one with EOB or EOF. If higher punctuation is found in a data stream where the datacomputer is looking for lower punctuation (e.g., an EOB where an EOR is expected), the higher punctuation implies the lower.

Interpretation: The interpretation of a STR is one of ASCII (i.e., 7-bit ASCII), ASCII8, or BYTE, as in the following examples:

```
A STR ASCII (5)
P STR ASCII8 (1,10)
WALDO STR BYTE (73)
```

The default byte size for BYTE is 36 bits. BYTE is optional if the byte size is given explicitly with the keyword parameter

,B=n

where n is a positive integer less than or equal to 36. The ,B=n option may not be used for ASCII STRings. If no byte size or interpretation is given, then the STR is 7-bit ASCII.

<u>Virtual Containers</u>: A virtual container is one whose value is not stored as data, but can be derived. The only kind of virtual container implemented in 0/11 is the virtual index container: the position, by number starting at 1, of a LIST-member within the LIST. For the virtual index container to be used, it must be included in the CREATing description with the following format:

<name> BYTE, V=I

The name is user-specified within the container ident restrictions; the data type must be BYTE and must be followed by ',V=I', which stands for 'virtual container equals index'. The virtual index container will take up no physical space within the file. Virtual index containers may exist in inner and/or outer STRUCT's.

<u>Fill Character:</u> At times the datacomputer needs to fill in a value or a part of a value. The user can specify a fill character thus:

,F='a'

or

,F=n

where a is an ASCII character and n is a decimal number representing a character code. The default fill character is a blank for ASCII data and zero for non-ASCII data.

Note that a byte size and a fill character can apply to a STRUCT or a LIST as well as a STR, INTEGER, or a BYTE. Consider the following:

```
CREATE F FILE LIST
     R STRUCT, B=36
       A STR (5)
     END;
```

The byte size of A is 7. A takes up 35 bits. There is one "unused" bit after A before the next R. Thus, R must be filled. Even though the data (i.e., A) is ASCII, R is non-ASCII because it does not have a 7-bit byte size.

Hence, the default filler of 0 is used for the bit.

The rules for punctuation, byte size and fillers are simple but not at all intuitive. In general, specifying punctuation rather than relying on defaults helps avoid errors. Also

```
LIST <pathname> %DESC;
```

will output a complete description, including all default lengths, dimensions, punctuation, byte sizes and fillers. (The LIST command is discussed more fully in Chapter 3.) It is often instructive to look closely at the %DESC to see where it is different from what is expected.

__Inversion and CAT:__ BYTEs, INTegers and STRings may be inverted. For members of outer LISTs, the option

```
,I=D
```

is used. For members of inner LISTs, the option

```
,I=I
```

is used. Inversions and the difference between outer list members and inner list members are discussed more fully in Chapter 5.

The Container Address Table (CAT) is a feature which can be used for retrieval of variable length data. The CAT is a table of pointers to the start of each variable-length LIST member. The CAT is automatically formed for those variable length LIST's which contain at least one inverted elementary data container. For example:

```
CREATE F FILE LIST
     A STR (,10), I=D;
```

The user can specify a CAT for LISTs which do not contain an inverted element. The CAT will be used to speed retrievals (discussed in Chapter 5).

```
CREATE F FILE LIST, CAT
     A STR (10)
     B BYTE, V=I;
```

__Byte Sizes__

Containers have a physical byte size. These byte sizes may be specified for FILEs for the purposes of packing data and for alignment of data. For PORTs, the user may specify byte sizes in order to model the transmission format of the

host machine. When byte sizes are not explicitly given, they have the following defaults: for STRs, the default is 7. For BYTEs and INTs the default is 36. For non-elementary containers the default byte size is that of the largest subcontainer. These defaults correspond to data being sent from a 36 bit machine with a seven bit character size -- for example a PDP-10. Byte size combinations of 8, 16, and 32 can be used to model data formats on 32 bit machines such as the IBM 360. In order to construct containers with byte sizes different from the defaults, the user must specify the byte size with the "B=n" option, where n is a decimal number of bits.

In version 0/11,

1. containers have a maximum byte size of 36 bits

2. no subcontainer byte can straddle 2 or more parent bytes

3. this implies that the byte size of the subcontainer must be less than or equal to that of the parent.

There are a number of temporary restrictions on BYTE size combinations. The rules are given in Appendix E, #8.

Chapter 3: Directory Commands

OPEN

Before data can be input to or read from a FILE or PORT, the container must be open, and a mode must be specified for it. The mode of a FILE or PORT, which is set when the container is opened, determines the legality of various operations on that container.

The possible MODEs are READ, WRITE, and APPEND. Data can be transmitted either out of or into a FILE or PORT that is open in WRITE or APPEND mode, but can only be transmitted out of a FILE or PORT that is open in READ mode. The difference between WRITE and APPEND lies in their treatment of any data that is already in the container when it is opened. When an assignment is made to a container that was opened in WRITE mode, any data it contained previously is thrown away. A container opened in APPEND mode has newly-arriving data written after the end of any already-present data, which is thus preserved.

A variation of WRITE and APPEND is WRITE DEFER and APPEND DEFER. When DEFER is indicated as part of the mode, a more efficient technique of building an inversion is used.

When a FILE or PORT is created, it is opened in WRITE mode. A FILE/PORT that already exists may be opened with an OPEN request:

```
OPEN <pathname> <mode> ;
```

which specifies the name of the container that is to be opened and the mode of opening. The name can be either a complete pathname (started with the reserved word %TOP) or it can be a login pathname, started with a node immediately subordinate to the current login node. The mode must be one for which the user has privileges (see Chapter 4). The mode argument may be left out of an OPEN statement, in which case the container is opened in READ mode if it is a FILE and WRITE mode if it is a PORT. Two outermost containers with the same ident may not be open at the same time.

For example, to read data that was previously stored in CCA.DATA.F, a file, either

```
OPEN %TOP.CCA.DATA.F;
```

or, if the current login node is CCA,

```
OPEN DATA.F;
```

will open F preparatory to data transfer requests.

MODE

The mode of a container that is already open may be changed with the MODE statement:

```
MODE <pathname> <mode> ;
```

The pathname can be a simple complete pathname (i.e. a complete pathname with no passwords), a simple login pathname, or a node name.

CLOSE

The complement of the OPEN request is the CLOSE request. When an open container is no longer needed, it can be closed with

```
CLOSE <pathname> ;
```

where pathname must be the simple pathname of an open container. Closing a FILE/PORT with a function of TEMPORARY PORT has the effect of deleting its description from the datacomputer.

DELETE

The ability to delete directory nodes is useful in maintaining a data base at the datacomputer. The DELETE request allows one to delete one or several outermost containers and all the data they contain.

```
DELETE <pathname> ;
```

causes the node named by <pathname> to be deleted from the directory. The pathname must be the login pathname. Thus, only nodes subordinate to the login node can be deleted. The node cannot have any subordinates.

```
DELETE <pathname>.** ;
```

deletes the node and all subordinate nodes. If any of the deleted nodes are outermost containers, the container descriptions and any associated data are deleted as well. The DELETE request need not be used on TEMPORARY PORTs, as they are automatically deleted either when they are closed,

or at session end.

If the data stored in a FILE is to be deleted, but the container description itself retained in storage, the DELETE request cannot be used. Instead, CREATE a port B with a description matching the container A that is to be emptied, and execute the assignment A = B with no data in B. The effect of this assignment is to delete all the data from A.

LIST

The LIST request is the means by which the user interrogates the datacomputer about his environment. The request has two arguments: the node or nodes which are the object of the inquiry, and the type of information desired.

The first argument consists of a set of nodes in the directory. Possible node sets are: 1) a single node, 2) all nodes directly subordinate to a given node, 3) a node and all its subordinates, and 4) all open files and ports. A single node is specified with a full pathname, which can include passwords and can be anchored at the top node (%TOP). The set of a node's direct subordinates is indicated with either a "*" (the login pathame is implicit) or a full pathname followed by a "*". Either "**" or a full pathname followed by a "**" designates a node and all its subordinates. The set of all open nodes is referenced by %OPEN. %TOP alone defaults to %TOP.**.

There are five kinds of available information. These are: 1) node names and related data (node type, privileges, and possibly mode and connected argument), 2) parsed data descriptions (of FILEs and PORTs), 3) original source text of data descriptions, 4) allocated space (for FILEs), and 5) privilege blocks associated with nodes. These information options are specified by %NAME, %DESC or %DESCRIPTION, %SOURCE, %ALLOC or %ALLOCATION, and %PRIV or %PRIVILEGE, respectively. The default option is %NAME.

Not all of the kinds of information are available for all of the possible node sets. The options that are available are:

Node Set	Option
<pathname>	%DESC
<pathname>	%NAME
<pathname>	%SOURCE
<pathname>	%ALLOC(ATION)
<pathname>	%PRIV(ILEGE)
<pathname>.*	%NAME
<pathname>.**	%NAME

<pathname>.**	%SOURCE
%OPEN	%NAME
%OPEN	%DESC(RIPTION)
%OPEN	%SOURCE
%OPEN	%ALLOC(ATION)

Chapter 4: Security and Passwords

Introductory Concepts

The 0/11 version of the datacomputer provides file-level security (restricted access to nodes and attendant data) by means of a system of privilege blocks, described in the following sections. One or more (or no) blocks may be associated with a particular node. Each privilege block defines a class of users who may be given access to the node and the set of privileges to be granted to such users. Whenever a user attempts to access a node or FILE/PORT, the datacomputer will scan that node or FILE/PORT's privilege block(s), if any, to ensure that the user is 'legal' and to determine what privileges will be allowed.

Chapter Organization

This chapter is divided into three principal parts. The first sections describe what privilege blocks are and how they provide file security functions for datacomputer users, and introduce the reader to the security features of datalanguage. The second part completely specifies the datalanguage needed for creating, deleting and manipulating privilege blocks, and completes the description of their components begun in the first part. The third section offers several examples of how to add, delete and look at privilege blocks.

Gaining Access to Nodes: LOGIN

Every node in the directory has certain privileges associated with it. For example, the ability to create inferior nodes, or to read or write file data, is a privilege which may be granted or denied to a particular node. When a user initially connects to the datacomputer he is automatically connected to the top node of the directory tree (%TOP), and he (i.e., the %TOP node) is granted minimal privileges. To acquire more, he must log in to some node, and this node is called the login node.

Logging into this node establishes the user's identity for subsequent pathname references (1). It should be kept in mind that a user is identified to the datacomputer only by his login node. Thus, throughout this chapter, the terms 'user-id' or 'user name' are to be understood to mean

- - - - - - - - - - - - - - - - - - - -

(1) In addition to establishing a user identity for privilege purposes, logging in performs various accounting and pathname context functions.

nothing more than the full pathname, including the specified privilege block (if any) at each level (2), of the node to which the user has logged-in.

Whenever a logged-in user references a node, the login pathname is compared against the user-id field of every block in the node's privilege block list. If a block is found whose user class description includes the pathname of the login node, the privilege-set described by the block will be added to (or taken away from) the privilege set already given to the login node.

Privileges

Privilege set specifications come in two flavors: privileges to be granted (added) to the node and privileges to be denied (taken away). If a privilege is not specified (as either grant or deny), then that privilege (or denial of it) is passed, unchanged, from the superior node to its subordinate. At each node level, the deny bits specified in the given privilege block are NOT-AND'ed with the privileges of the superior node. Then the grant privileges are OR'ed with the result, to yield the privilege set for that node.

It is important to understand that privileges may be added and taken away at every level of the pathname. For example, suppose the login node has the privilege set <CLWA> (3), and a subnode's privilege block specifies: grant read privilege (G=R), and deny write privilege (D=W). The result at the subnode would be the final privilege set of <CRA> (4).

- -

(2) Pathnames may be qualified or unqualified. A qualified pathname is one containing password strings for the purpose of gaining particular privileges upon opening the node, e.g.,

NODE1('PASSWORD1').NODE2.NODE3('PW3')

is a pathname qualified at the first and third levels by the passwords 'PASSWORD1' and 'PW3', respectively. The pathname NODE1.NODE2.NODE3, on the other hand, is unqualified. Prior to Version 0/10, all pathnames were unqualified.

(3) This is a shorthand way of saying 'this node has been granted control <C>, login <L>, write-to-file <W> and append-to-file <A> privileges.' Specific privileges are described in detail below.

(4) The login privilege is not propagated to subnodes. It applies only to the node for which it is explicitly granted. See below.

Note that a node's privilege set can never allow a user look at, modify, or affect a superior node in any way not possible at the level of the superior. That is, if a user cannot look at the privilege blocks for a node, he cannot acquire that privilege for that node from an inferior one. However, an inferior node may well have privileges relative to its subnodes that its superior does not have relative to its subnodes. For example, scanning along the pathname A.B.C.D.E...., A.B.C may have only read privileges, but does not have rite privilege. Now, the node A.B.C.D may be granted write privilege at level D (thus awarding A.B.C.D read/write privileges), this does not affect A.B.C. It still has only read privilege.

Privilege Block

Privilege blocks are data structures which define access to nodes. Each privilege block is associated with one particular node. Any node in the directory, including PORTs and FILEs, may have privilege blocks defined for it. A node may have any number (including zero) of privilege blocks. When an attempt is made to access a node which has privilege block(s), those blocks are scanned for a user-id corresponding to the current login pathname and for a password string matching that supplied by the user in the request referencing the node (e.g., LOGIN, OPEN, DELETE, etc.). If a match is found, the matching block's privilege bits are examined and the appropriate privileges are granted/denied the node. The matching algorithm is described below in more detail.

Each privilege block can contain:

- user name
- host name
- socket number
- password character string
- grant privileges
- deny privileges

Each of the above fields falls into one of two categories: 1) a description of the group of users which may access the associated node; and 2) the privileges to be granted to these users.

The privilege block is completely specified at the time it is created. When a node is referenced, only the password string, if any, is required; the user-id (including host name and socket number), has been retained by the login process.

Privilege blocks are created by the datalanguage command CREATEP. They are deleted by the command DELETEP. Existing privilege blocks may be displayed via the LIST nodename

$PRIV(ILEGE) command. The full syntax of these commands is described below.

User Identification Fields (User-ID)

The user identification fields include some or all of the following: a valid login pathname or a class of login pathnames, the number of a host computer, the foreign host socket number, and a password character string. These fields are discussed in more detail in the following sections.

Host

The host name is an optional field. If specified, it must be a decimal number from 1 to 255 designating the number of the host computer. The host name cannot be a number greater than 255, or less than 1. It cannot be a character string, except for the special cases LOCAL and ANY.

The host name may also be ANY, which means that any host, foreign or local, is acceptable.

If a host name is not specified, the default value is ANY.

User Name

The user name is the pathname or classname (5) of the login node(s) which may gain access to the node associated with the privilege block. Note that a different privilege block must be created for each specific user permitted to use a given password. For example, if two different users, say CCA.WALDO and CCA.DINGLE, wanted to use the same password string ('FOO') to gain access to a node, two separate blocks would have to be created, one per specific user name. Thus, in this example, one privilege block would contain the information

    CCA.WALDO ('FOO');

the other,

    CCA.DINGLE ('FOO').

If no user name is specified, the default is **, which grants any user access to the node.

- -

(5) User classnames are defined below.

Socket

The socket number is a 32-bit decimal number, e.g., 609403, or ANY. This is an identification number assigned by the foreign host to the user logged in on that foreign host. Usage of the socket number in the CREATEP statement can ensure, for TENEX systems, that only specified users at the foreign host site may gain access to a particular node.

Socket number defaults to ANY.

Password

A password consists of an alphanumeric string enclosed by single quote (') characters, e.g., P='FOO'. Non-printing characters, except blanks, are not valid in a password string. Blanks may appear at any point in the quoted string. Tab characters are not permitted.

A privilege block need not contain a password. If it does not, none should be given when referencing that node. Note that _no_ password is _not the same as_, and is treated differently from, a _null_ password (''). The null password is treated as a password of zero length, and must be supplied as such whenever the node is referenced.

Privilege set specifications

The following privilege bits are defined for 0/11:

LOGIN (L) In order to control login identities more closely, the ability to log in to a node is not passed to subordinates. As a result, -L (deny login) is meaningless.

CONTROL (C) Control includes complete subordinate control and privilege control. Control is required for creating and deleting nodes, files and privilege blocks. It is also required for listing privilege blocks. It is very powerful, and cannot be removed by an inferior: -C is not permitted. After 0/11, C may be split into meaningful components

Data Control Privileges
READ (R)
WRITE (W) W implies R and A.
APPEND (A) A does not imply R.
Conflicts are not allowed, e.g. +R and -R.

Ordering of Privilege Blocks

__The ordering of privilege blocks is important.__ When a node is referenced, the privilege blocks (if any) for that node are scanned linearly for a password string matching the password entered by the user. If a match is found, the user-id of the privilege block is compared to the login identity. If they match, the associated privileges are granted/denied, and access appropriate to the granted privilege set are awarded to the node. If the end of the privilege blocks is reached without finding a password/user-id match, the node is opened with no privileges.

Since the privilege blocks are scanned linearly, their ordering defines their selectivity. F^r example, suppose a node to have two privilege blocks which specify the same password ('FOO') but different login nodes, say, A and ** **, and suppose that the block with user name A grants greater privileges (read/write/append) than that with ** (which permits read). The proper ordering, as displayed by a

```
LIST WALDO.NODENAME $PRIV(ILEGE);
```

statement, is as follows: (6)

```
(1),U=A,H=ANY,S=ANY,G=RWA

(2),U=**,H=ANY,S=ANY,G=R       (7)
```

If the order of these blocks were reversed, so that the block with the user name '**' were first, then whenever the password FOO was encountered the first block would be selected; i.e., every login pathname would match the '**', and the matching process would be complete. Thus, the block with the user name A would never be found, and the user A would be unable to open the node with the greater privileges which should be granted him.

In 0/11 the user is responsible for maintaining the desired search order, by adding and deleting privilege blocks via their block numbers. The datalanguage for this process is described below. Future versions of the datacomputer may

\- - - - - - - - - - - - - - - - -

(6) Details of this command are given below.

(7) U=** means that any user name will be accepted as valid.

provide an automatic ordering algorithm, which could be manually overridden, if desired.

<u>User Classes ('Star' Feature)</u>

Classes of users may be given access to a node by specifying a user class as the user name instead of a single user. This is done by means of the '*' and '**' ('star' and 'star-star') features. If a star appears in a pathname, it is interpreted to mean: 'any single (non-null) partial pathname is acceptable here'. That is, if the nodes A.B.N1, A.B.N2, and A.B.N3 exist in the directory tree, usage of the user classname A.B.* would specify any of these three pathnames. Stars may appear at any number of levels; for example, if the nodes A.X.N1 and A.Y.N4 exist, then the user-name A.*.* would specify both of these nodes, as well as any of the previous three. The use of a star at any level implies that there must be a partial pathname at that level; e.g., the classname A.*.* could not specify node A or A.J.

<u>User Classes, cont. ('Star-star' Feature)</u>

The use of a single star in a pathname indicates that a node must exist at the level corresponding to that of the star, and a star must be explicitly specified for each desired level. The star-star feature is designed to permit access to several levels of nodes. A star-star ('**') in a user name is interpreted to mean: 'any number (including zero) of partial pathnames are acceptable here'. Thus, referring to the example of the preceeding paragraph, A.B.N1 could be specified by any of the following:

```
A.B.N1
A.B.N1.**
A.B.*.**
A.B.**
A.B.*
A.*.**
A.**
*.**
**
```

For 0/11, only trailing *'s and/or a final ** are allowed. The following, for example, are illegal:

```
A.*.C
A.**.C
A.*.**.D
A.**.*
*.B.**
**.*
```

Datalanguage for File Security

Two datalanguage statements, CREATEP and DELETEP, create and delete privilege blocks. They are discussed in the following sections. The LIST command has a special option, $PRIV (or $PRIVILEGE), which allows the user to list the privilege blocks for a node.

CREATEP and DELETEP are privileged requests. They are accepted only when the associated node can be referenced with control privilege <C>. (This means that it may be necessary to login to some particular node before any privilege blocks can be added to another, and that passwords may be required for the login process or for referencing nodes superior to the node for which the privilege block is to be added.)

Creating Privilege Blocks: CREATEP

Privilege blocks are created, and fully specified by, the CREATEP command. A fully specified CREATEP statement might appear as follows:

```
CREATEP NODE1('PW1').NODE2, U=CCA.WALDO.*.**, H=34,
                         S=604320,
        P='SECRET PASSWORD', G=R, D=WA, N=2;
```

In this example, the node for which we are creating a privilege block is NODE1.NODE2. We must specify ('PW1') for NODE1 in order, perhaps, to gain control privileges at the first level. The parameters which follow the nodename are the privilege keyword list. These are discussed individually in the following sections, and are summarized in Appendix A.

CREATEP: User Name

The user name is specified by 'U=' followed by an unqualified pathname or classname string. The pathname may have any number of levels. It must not contain password strings for any level.

The following are valid pathnames/classnames.

```
CCA
CCA.WALDO.DINGLE
CCA.*.*
CCA.**
*.*.*
*.**
**
```

CREATEP: Host Number

The host number is specified by 'H=' followed by a decimal number from 1 to 255, or either of the strings LOCAL or ANY.

```
H=28
H=ANY
H=LOCAL
```

CREATEP: Socket Number

The socket number is specified by 'S=' followed by the 32-bit foreign-host assigned decimal number corresponding to the directory the user is logged into at that foreign host, or the string ANY.

```
S=309483
S=ANY
```

CREATEP: Password String

The password string is specified by 'P=' followed by any datacomputer string constant (tabs may not be included, although blanks are permitted), e.g., 'PASSWORD 1', '? * ++!!', or '' (null password).

Note that if no password string is specified at CREATEP time, then that privilege block will have no password associated with it. No password is different from null password (P=''), which is a valid password zero characters in length.

CREATEP: Grant Privileges

Privileges are granted by 'G=' followed by

- C (control)
- L (login)
- R (read FILE/PORT data)
- W (write FILE/PORT data)
- A (append data to FILE/PORT)

in any combination and in any order, e.g., G=CRAWL (all privileges), G=WAR (read/write/append), etc.

CREATEP: Deny Privileges

Deny privileges are specified by 'D=' followed by R, W or A. Login (L) applies only to the node for which it is specified. It is not passed to subordinates. Control (C) cannot be removed by any inferior node, i.e., it is passed to all subnodes.

CREATEP: Privilege Block Number

As privilege blocks are created, they are assigned numbers by the datacomputer. Block numbers are assigned to privilege blocks sequentially according to their search order. Block numbers can range from one to n, where n is the total number of password blocks in the search sequence. Blocks can be explicitly ordered by the user at CREATEP time by entering 'N=' followed by the number that the newly added block is to have in the search sequence. N must be greater than zero, and not greater than the total number of privilege blocks currently existing for the node. Note that this number is not in any sense a part of the data contained in the privilege block; it is merely the position of the block in the password block list.

An example. If there were three blocks in the privilege block list for a node (NODE1),

```
1  U=AAA
2  U=CCC
3  U=DDD
```

and a new block were to be added between the first and second existing blocks, i.e., so that the new block would then occupy second position, we add a keyword, N=2, to a CREATEP command:

```
CREATEP  NODE1,U=BBB,P='ZOO',N=2;
```

which results in the following privilege block list:

```
1  U=AAA
2  U=BBB
3  U=CCC
4  U=DDD
```

If N had been omitted, the new block would have been added at the end of the list. Note that the numbers of the two blocks following the new one have been bumped by one. Similarly, if any block is deleted, the numbers of all the following blocks are reduced by one.

LOOKING AT PRIVILEGE BLOCKS: LIST

In order to permit the user to list privilege block information, the %PRIV (or %PRIVILEGE) option exists for the datalanguage LIST request. It looks like this:

```
LIST CCA.WALDO  %PRIV;      (or)
LIST CCA.WALDO  %PRIVILEGE;
```

Passwords cannot be listed with the $PRIV option (or in any other way - so don't forget them!). Privilege block information is preceded by the number of that block. All other information in the privilege block is listed in a format similar to that which might be found in a CREATEP command, e.g, either of the LIST requests above might generate the following output from the datacomputer:

```
(1),U=CCA.WALDO,H=LOCAL,S=ANY,G=CRAWL
(2),U=CCA.*.**,H=ANY,S=ANY,G=RWAL
(3),U=*.**,H=32,S=654364,G=RL,D=WA
```

$PRIV may be used only when the node has control privileges.

Deleting Privilege Blocks: DELETEP

Privilege blocks may be deleted with DELETEP followed by the number of the privilege block to be deleted,

```
DELETEP NODE1, 3
```

The controlling node must have control privilege.

Example

This example will create a node which will be the controlling node for all other nodes at site CCA. Presumably, access to this controlling node would be restricted to very few persons at that site; 'super-users', as it were. This could be done by means of a password. In addition, anyone seeking control privileges for CCA might be required to be logged-in to some other (access restricted) node. The person with access to CCA would be responsible for creating subnodes, perhaps one for each programmer permitted to use the datacomputer. These individual programmers could then create their own directory structures (nodes, ports and files) in any manner they wish.

The site-node CCA is created by the following series of requests:

```
CREATE CCA;
CREATEP CCA,P='HONCHO',G=CL;
CREATEP CCA,P='FLUNKY',G=L;
LOGIN CCA('HONCHO');
```

The user is now logged in to CCA. He has control privileges. Next he creates a series of programmer-nodes,

each with control privileges. Initially, two privilege blocks are created for each programmer node. One requires a password (known to, and probably specified by, the individual programmer), and the other requires no password and is accessible to anyone logged in to CCA or any of its subnodes. However, persons who log in to a programmer node without specifying a password are not given control privileges and thus cannot modify or delete anything that the programmer wishes to keep secure.

```
CREATE WALDO;    CREATEP WALDO,U=CCA,P='TURKEY',G=CL;
                 CREATEP WALDO,U=CCA.**,G=L;
CREATE CLYDE;    CREATEP CLYDE,U=CCA,P='FETCH',G=CL;
                 CREATEP CLYDE,U=CCA.**,G=L;
      .
      .
      .
      .
CREATE DINK;     CREATEP DINK,U=CCA,P='PODUNK',G=CL;
                 CREATEP DINK,U=CCA.**,G=L;
```

After this is done, super-user checks the privilege blocks he has created, first at his own node level:

```
LIST $TOP.CCA('HONCHO') $PRIVILEGE;
```

and he receives a datacomputer printout in the following format:

```
(1),U=**,H=ANY,S=ANY,G=CL
(2),U=**,H=ANY,S=ANY,G=L
```

He next verifies that each of the programmer-node privilege blocks has been correctly entered, e.g.,

```
LIST WALDO $PRIV;
```

and the datacomputer replies:

```
(1),U=CCA,H=ANY,S=ANY,G=CL
(2),U=CCA.**,H=ANY,S=ANY,G=L
```

At this point, programmer Waldo tells super-user that he would rather have 'DONKEY' as his control password rather than 'TURKEY'. Since the user name (U=CCA) in Waldo's control privilege block is more restrictive than the user name (U=CCA.**) in the non-control privilege block, the first privilege block must be deleted and the new one added in the same position (N=1):

```
DELETEP WALDO 1;
CREATEP WALDO,U=CCA,P='DONKEY',G=CL,N=1;
```

We now have the following directory:

```
CCA
CCA.WALDO
CCA.CLYDE
    .
    .
    .
CCA.DINK
```

Each of the programmer-nodes listed above has its own password which is known to the person having access to that node. In addition, each is required to login to CCA before being able to acquire login and control privileges at its own level. (Most or all of the programmers at CCA are given only the password FLUNKY, which does not give control privileges. Thus, they cannot create or delete any nodes at the programmer-node level or look at the restricted data of any other programmers.)

As soon as he is informed that he may join the select international hoard of datacomputer users, Waldo rushes to his terminal to login:

```
LOGIN CCA('FLUNKY');
LOGIN WALDO('DONKEY');
```

Since he has logged in to his node using the password which grants control privileges, Waldo now creates BOOKFILE and BOOKPORT and reads some data into BOOKFILE from a TENEX file named TENEX-BOOK.FILE (8):

```
CREATE BOOKFILE FILE LIST(,1000)
  BOOK STRUCT
        TITLE STR (,100),C=1
        AUTHORS LIST(,5),C=1
          AUTHOR STR (,50),C=1
        PUBLISHER STR (,50),C=1
  END;

CREATE BOOKPORT PORT LIST(,1000),P=EOF
  BOOK STRUCT
        TITLE STR (,100),P=EOR
        AUTHORS LIST(,5),P=EOB
          AUTHOR STR (,50),P=EOR
        PUBLISHER STR (,50),P=EOR
  END;

CLOSE %OPEN;

OPEN BOOKFILE WRITE;
```

```
OPEN BOOKPORT;
CONNECT BOOKPORT 'TENEX-BOOK.FILE';   (8)
BOOKFILE=BOOKPORT;

CLOSE $OPEN;
```

In order to permit others to look at his BOOKPORT, Waldo creates a couple of privilege blocks. The first permits anyone at CCA to look at his book list, while denying him the right to change anything. The second is for Waldo's private use in changing the file:

```
CREATEP BOOKFILE,U=CCA.*,G=R,D=AW;

CREATEP
 BOOKFILE,U=CCA.WALDO,P='READ*MORE*EVERY*DAY',G=RWA;
```

- - - - - - - - - - - - - - - - - - - -

(8) A TENEX filename is used in this example for the purpose of didactic clarity. In practice, this would usually be done only by local datacomputer users (users located at the site of the datacomputer). Remote users would have to arrange for operator intervention, if connecting to a TENEX file at the datacomputer site; or would specify the host name and socket number from which the data would be sent to the datacomputer.

Chapter 5: Assignment and For-loops

Assignment Involving Outermost Containers

Transmission of data is achieved with an assignment. The syntax of an assignment request that involves two outermost containers is

```
<ident> = <ident>;
```

where the <ident>s are the node names of open outermost containers. The first ident in the statement is that of the receiving container; it must be open in either WRITE or APPEND mode. The second ident is that of the transmitting container; it can be open in any mode, but it must have READ privilege (see Chapter 4).

The containers in the assignment may be either files or ports. The various combinations are listed here, with a description of the action of the assignment request in each case.

Receiving container	Transmitting container	Comment
FILE	FILE	copies data from one FILE to another within the datacomputer.
FILE	PORT	transmits data from some source external to the datacomputer through a PORT, into a FILE.
PORT	FILE	transmits data from a FILE, where it is being kept in the datacomputer, through a PORT, to the outside world.
PORT	PORT	transmits data from one place to another in the outside world, using the datacomputer only as a channel for transmission.

The Matching Rules

In any assignment statement such as

```
X = Y;
```

(not only one involving two outermost containers) the two operands, X and Y, each have their own description. The datacomputer will transform the data in Y to match the description of X. In order for the datacomputer to be able to do this, the descriptions must match. This amounts to a restriction that only similar objects can be assigned to each other. Specifically, for two assignment-operands X and Y to match:

1.A. X and Y must have the same type: LIST, STRUCT, or data types STR, INT, or BYTE,

AND

1.B. If X and Y are both LISTs, then they must have compatible sizes, or else X must be a PORT. The sizes are compatible if the minimum size of X is less than or equal to the minimum of Y and the maximum size of X is greater than or equal to the maximum size of Y. This restriction leads to cases where it is legal to assign Y to X but not to assign X to Y. Note that if X and Y are outermost lists with no list size specified, the datacomputer supplies a default size based on the space allocation. (Use the LIST request with the %DESC option to find out what the default size is.)

AND

1.C. If X and Y are STRUCTs or LISTs, then at least one container immediately enclosed in X must match, and have the same ident as, one container immediately contained in Y,

OR

2. X must be a STRing, INTeger or BYTE and Y a constant. A constant is an arbitrary string of characters. If they are enclosed by single quote marks, then it is an ASCII constant. If they are not inclosed by quote marks, then the string is used as a binary constant; a single or double quote mark may be included in such a string only by prefixing it with another double quote. The constant 'DON"'T' represents the string DON'T. (This rule is included here for completeness.)

Padding and Truncation

If two containers of type STR are used in an assignment, the matching rules do not require that their sizes match. There are three cases:

1. The two sizes are equal. The string is assigned without change.

2. In the assignment X=Y, the size of X is greater than that of Y. In this case, it is as if the string in Y is padded at the right-hand side to make it as long as X, before assignment is performed. If a fill character is specified in the description of X (i.e. if the parameter

,F='a' or ,F=n is used in the CREATE request), then that character is used. Otherwise, a blank is used for ASCII strings and zero is used for non-ASCII data.

3. The size of X is less than that of Y. The string contained in Y is truncated at the right-hand side to be as short as X, and the shortened string is then assigned.

Conversion

It is possible to assign the data type STR, if it contains a number, to the data types BYTE and INT, and the reverse. In such an assignment, the input ASCII STR can be any length, but its binary magnitude must fit in 35 bits, or the specified size of the BYTE or INT. The only legal characters are numbers, + and - signs, and blanks. Leading and trailing blanks are allowed but embedded blanks are not. Any number of sign characters may precede the number, mixed with blanks; even numbers of minus signs cancel. A string with no digits is an error; any error causes a zero result. An input BYTE is treated as a positive number regardless of the high order bit. If the input (right hand of the assignment statement) INT or BYTE will not fit properly in the output bytesize, an error statement is made, and the result is truncated on the left. An output ASCII STR will contain no more than eleven significant digits. If the number being converted for output is negative, a minus sign will appear in the first character position in the output string. The minumum possible number of digits will be output, with leading zeros only if the STRs minimum length is greater than eleven. If the STR does not contain enough positions to hold the entire number, an error statement will be made.

Examples

Let us consider a few examples of the operation of the rules. Suppose we have

```
CREATE M FILE LIST (25)    RECORD STR(10);
CREATE N TEMP PORT LIST (25), P=EOF RECORD STR(10) ;
M = N;
```

where M is a FILE in which data read from the PORT N is to be stored in the datacomputer. The assignment M = N is legal because M is in WRITE mode and both M and N are open (opened by the CREATE statements and the MODEs set). In addition, M and N match: their subcontainers have the same ident (RECORD), and matching descriptions. They satisfy rule 1.A, since the type is STR in both cases, and rules 1.B and 1.C do not apply to containers of type STR.

The effect of this assignment is to read strings of

length 10 from the PORT N, and to store them in the FILE M. If an attempt is made to store more than 25 strings in M, an error message is output, as space was allocated for only 25 strings.

A similar example, using the above description for M:

```
    OPEN M APPEND;
    CREATE O TEMP PORT LIST (25), P=EOF
             RECORD STR (,15), P=EOR ;
    M = O;
```

Each STRing in O is no more than 15 ASCII characters and ends with an EOR. Each one will be padded or truncated to 10 characters since M has fixed-length rather than variable length STRings.

Now a more complex example.

```
CREATE FF FILE LIST (,25)
     PERSON STRUCT
        NAME STR (15)
        ADDRESS STR (20)
        CITY STR (10)
        STATE STR (2)
        ZIP STR (5)
        SOCSECNO STR (9)
        DEPENDENTS LIST (10) NAME STR (15)
     END ;
... requests that store data in the FILE FF ...
CREATE PP PORT LIST, P=EOF
    PERSON STRUCT, P=EOR
        NAME STR (15)
        SOCSECNO STR (9)
    END;
   PP = FF ;
```

Here, the assignment PP = FF is legal because: PP is in WRITE mode, both FF and PP are open, and their descriptions match. Rule 1.A: the type of both FF and PP is LIST. Rule 1.B: PP is a PORT. Rule 1.C: the subcontainer PERSON immediately contained in FF has the same ident as the subcontainer PERSON in PP, and the two STRUCTs PERSON match. We determine this last fact by going around once again with the matching rules.

Rule 1.A: PERSON in FF and PERSON in PP have the same type, STRUCT. Rule 1.B does not apply to STRUCTs. Rule 1.C: a container immediately contained by PERSON in FF, NAME, has the same ident (NAME) and a matching description (STR (15)) as a container immediately enclosed by PERSON in PP, that is, NAME.

The effect of this assignment is to create a new instance of the struct PERSON for each instance of PERSON in

FF, and add it to the LIST PP (that is, output it through the PORT PP). Each PERSON that is output contains only a selection of the data stored in FF: only the NAME and SOCSECNO.

If the situation here were reversed, that is, if FF were open in WRITE mode, and PP were in READ mode, the effect of the assignment

```
FF = PP;
```

would be to read data from the PORT PP and store it in the FILE FF. However, only the NAME and SOCSECNO would be available as data. The datacomputer handles this situation by assigning strings consisting only of blanks (the default since no fill character is specified in the description) to the unmatched STRs in the output LIST-member. Thus, ADDRESS, CITY, STATE, ZIP, and all 10 instances of NAME in the DEPENDENTS LIST would be blank in the FILE FF.

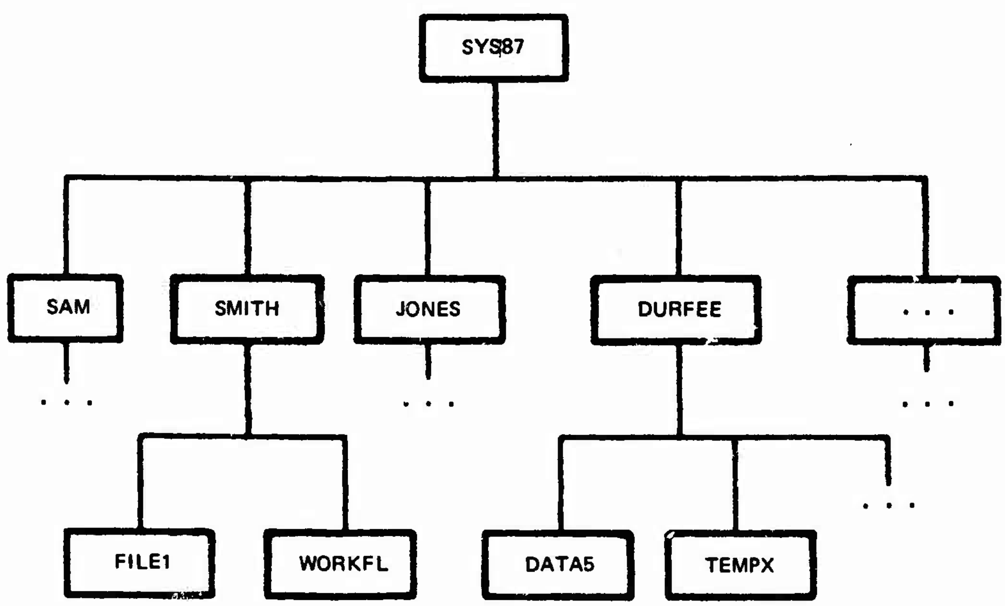

Figure 5-3. The directory for a sample application: providing backup file storage for time-sharing users

A directory of this sort would initially be set up by several CREATE requests; i.e.

```
CREATE SYS87;
  CREATE SYS87.SAM; CREATE SYS87.SMITH;
  CREATE SYS87.JONES; etc.
```

Then, whenever a particular file was to be moved to the datacomputer, a directory node for that file would be set up by, for example,

```
        CREATE SYS87.SMITH.FILE1 FILE LIST (,999)
                     A STR(80);
```

(describing a file with less than 1000 80-character records) and the file would be moved with an assignment statement specifying a PORT with a matching description, and the FILE FILE1, open in WRITE mode. Thus:

```
        CREATE T TEMP PORT LIST A STR(80);
          FILE1 = T;
```

Note that the two outermost containers FILE1 and T in the assignment statement FILE1 = T match each other.

In order to recover the file from the datacomputer when it is again needed, a PORT would be opened in WRITE mode with

```
       CREATE T TEMP PORT LIST A STR(80);
        OPEN SYS87.SMITH.FILE1 READ;
       T = FILE1 ;
```

and the reverse assignment would take place.

<u>Selection of LIST Members</u>

In the examples given above, there is one output LIST member for every input LIST member. Subsets of the input LIST member (i.e. the LIST on the right side of the =) may be specified by the use of a WITH clause. For example, consider the description

```
          CREATE F FILE LIST
                  P STRUCT A STR(3) B STR(5) END;
```

and a matching PORT R. If only some of the P's on the LIST F were to be output -- those with the string A equal to the string '500', say -- one could specify

```
       R = F WITH A EQ '500';
```

referring to the set of all members P of the LIST F that have the given property. Note that A is understood to refer to F.P.A (STR A in STRUCT P of the outermost container FILE F); see the section on the context rules below for an explanation. Quotes are used in the expression '500' to

indicate that an ASCII string constant is intended.

In a WITH clause, the expressions one can use to choose certain LIST-members, which are called Boolean expressions, must involve comparison of a container that is a STR, INT, or BYTE with a constant (like '500' in the example), using the comparison operators

```
      EQ  (equals)
      NE  (not equal to)
      GT  (greater than)
      LT  (less than)
      GE  (greater than or equal to)
and   LE  (less than or equal to).
```

Combinations of comparisons with

```
OR, AND, NOT, and ANY
```

are also possible. In precedence of operators, ANY (see below) is highest; NOT is next in precedence, then AND, which is in turn higher than OR; parentheses may be used to affect the order of evaluation of these operators. Some sample input-specs are thus:

```
F
F WITH A EQ '500'
F WITH A EQ '500' AND B GT 'AZZZZ'
F WITH (A EQ '500' AND NOT B GT 'MONDA') OR
         (A EQ '600' AND B NE 'ZYYYY')
```

For ASCII containers, the operators GT, LT, etc. compare the ASCII codes for the given strings and the given strings must be of the specified length. This means that the character blank is less than the digits, which in turn are less than the letters. Consult a reference document for the complete list of ASCII codes for all characters.

Also, while an input-spec like

```
F WITH A EQ '5'
```

is legal, it will not find any P's, since there are no A's with only one character.

Data will be compared to other data or constants without reference to interpretation as STR, INT or BYTE. Care should be taken to specify quoted ASCII constants for ASCII STRs and unquoted integer constants for INTs and BYTEs. A warning message will appear if the operands in a comparison are of different types, but the operation will continue.

Retrievals Using Inner List Members

Consider a description like

```
G FILE LIST
  R STRUCT
         A STR (4)
         B STR (4)
         W LIST (20)
          WA STR (5)
END
```

Each R has 20 WA's, since R contains an inner list (W). An input-spec like

```
G WITH WA EQ 'ABCDE'
```

specifies all R's with at least one WA with value 'ABCDE'. This may also be expressed as

```
G WITH ANY WA EQ 'ABCDE'
```

The former is called an _implicit ANY_ and the latter, an _explicit ANY_.

The container WA can be used in boolean expressions such as

```
G WITH ANY (WA EQ 'MARCH' OR WA EQ '33103')

G WITH ANY (WA EQ 'MARCH' OR WA EQ 'WORD ')

G WITH ANY WA EQ '12345' AND B EQ 'CALI'
```

An ANY expression _cannot_ be used within the object of another ANY expression (nested ANY's).

In most cases, the explicit ANY is not required. However, consider the description:

```
FAMILIES  FILE LIST (,100)
  FAMILY STRUCT
        MOTHER STR (10)
        FATHER STR (10)
        CHILDREN LIST (10)
                CHILD STRUCT
                  NAME STR (,10), C=1
                  AGE STR (2)
                END
END;
```

The following expressions are not equivalent:

```
FAMILY WITH ANY (NAME EQ 'ELLEN' AND
   AGE EQ '21')
FAMILY WITH NAME EQ 'ELLEN' AND
   AGE EQ '21'.
```

The latter case is interpreted as:

```
FAMILY WITH ANY NAME EQ 'ELLEN'
   AND ANY AGE EQ '21'
```

and refers to any FAMILY with an ELLEN who either is 21 or has a sibling who is 21. The former refers only to FAMILYs with a 21-year-old ELLEN.

In all of these examples, the inner list is the second-level list. If there is a third level list, its members may not be used in a boolean expression. For example, given the description:

```
F FILE LIST R STRUCT
        A STR(1)
        L LIST (5)
                L1 LIST (5)
                        B STR (1)
END;
```

L1 is a third-level list, and so B cannot be used in a WITH expression. However, A may still be used in a WITH expression.

Retrievals Using Inverted Containers

An elementary data container may be _inverted_ if it is contained in a FILE which is a LIST. This is useful if the container will be used often in a boolean expression. Inversion is specified by "I=D" or "I=I" as follows:

```
CREATE F FILE LIST (0,100)
  P STRUCT
   A STR (3), I=D
   Q LIST (10)
     B STR (5), I=I
  END;
```

The "I" of the above stands for inversion, the "=D" (for distinct) is used with members of outer lists, the "=I" (for indistinct) with inner lists.

An inversion on the string A increases the efficiency of retrieving sets of outermost-LIST members by the contents

of the string A -- that is, retrieving subsets of the P's that are defined by their values of A. Retrieval by content based on a particular string is possible whether or not that string is inverted; only the efficiency is improved by the existence of an inversion on the string.

There is a certain cost associated with inversion, however. Storage space must be allocated for a secondary data structure that the datacomputer uses for retrievals based on inverted strings. Appending to a FILE takes longer when it is inverted, since the secondary data structure must be changed as well. Thus, the decision to invert a particular string will depend on the relative cost of increased retrieval time versus increased storage space, the frequency of retrieval based on the particular string, and other considerations. Appendix C contains further technical details concerning inversion.

Retrievals Using Container Address Table. (CAT)

The Container Address Table (CAT) is a feature which will speed retrieval of variable length data. The CAT is a table of pointers to the start of each variable-length LIST member. The CAT is automatically formed for those variable length LIST's which contain at least one inverted container. For example:

```
CREATE F FILE LIST
     A STR (,10), I=D;
```

The user can specify a CAT for LIST's which do not contain an inverted container. The CAT will be used for retrievals based on the virtual index container for FILES of a format similar to the following:

```
CREATE F1 FILE LIST (,50000), CAT
   A STRUCT
    B STR (,10)
    C STR (5)
    X BYTE, V=I
   END;
```

Retrievals would be of the form (P1 a matching PORT):

```
P1=F1 WITH X EQ 2413;
```

or

```
P1=F1 WITH B GT '1000';
```

These retrievals can be executed whether or not there is a CAT, but the execution is faster with a CAT. There is, of course, a storage cost to the CAT, which is proportional to the number of LIST members.

Assignment with FOR

Containers other than outermost can also be used in assignment statements, if they are inside a FOR loop. FOR causes some set of datalanguage statements (usually assignment statements) to be executed several times, once for each member of a given set of LIST-members.

The syntax of the FOR-request is:

```
FOR <output-spec>, <input-spec> <body> END ;
```

The <input-spec> specifies a set of LIST-members to which the operations specified in the <body> are to be applied. A new member of the LIST specified by the <output-spec> is created for each member of the input set processed. If the output-spec is omitted, the FOR-request generates no output.

The input-spec The input-spec must specify a set of LIST-members. The simplest kind of input-spec is an entire LIST -- i.e. the set of all the LIST-members. For example, if

```
CREATE F FILE LIST
         P STRUCT  A STR (3)  B STR (5)  END;
```

then F would be a legal input-spec, and would refer to the set of all P's in the LIST F (1).

A subset of the LIST-members may be specified by the use of a WITH clause in the input-spec. The input-spec on a FOR-loop looks like the input spec on the assignment of outermost containers (discussed above). Thus

```
F WITH A EQ '500'
```

can be used in a FOR-loop.

The output-spec The output-spec is an optional argument. Like the input-spec, it must be the name of a LIST-member. The LIST that contains the LIST-member specified by the output-spec is often called the output LIST. A new member is created and added to the output LIST for each execution of the FOR-body.

A FOR-loop may be loosely thought of as assignment between two LISTs. However, the descriptions of the members of the input and output LISTs need not match. Otherwise,

- -

(1) Note the syntactic difference from version 0/10, LIST (instead of LIST member) naming for loop arguments.

the restrictions governing the input and output LISTs of a FOR are largely the same as those governing outermost LISTs used in assignment:

1. Both LISTs must be open or contained in open outermost containers.
2. The output LIST or its outermost container must be in WRITE or APPEND mode.
3. If the input LIST is not an outermost container, the LIST that most immediately encloses it must be the input LIST of an enclosing FOR loop.
4. Similarly, if the output LIST is not outermost, the LIST that most immediately encloses it must be the output LIST of an enclosing FOR.

<u>The FOR-body</u> The operations that are legal in a FOR-body are assignment and another (nested) FOR. The assignment may be of the form

```
&lt;name&gt; = &lt;constant&gt; ;
```

where <name> refers to a container that is a STR, INT or BYTE (see matching rule number 2, page 33), or assignment may be of the form

```
&lt;name&gt; = &lt;name&gt;;
```

to transfer data from one container to another. If the latter is the case, then assignment is subject to

1. the restrictions specified in the matching rules above,
2. the usual restriction that data can be transmitted into a container only if it is open in WRITE or APPEND mode, and
3. the restriction that assignment must occur between <u>objects</u>, not <u>sets of objects</u>.
4. In Version 0/11 of datalanguage, there are other restrictions governing the containers that can be referenced in the body of a FOR-loop. See Appendix E.

Let us look at a few examples, and describe their operation in words. With F a FILE as above, and

```
CREATE Q PORT LIST
    P STRUCT
        A STR (3)
        B STR (5)
    END;
...
OPEN F APPEND;
```

then

```
F = Q;
```

and

```
FOR F, Q
```

```
          P = P ;
      END;
```

have the same effect: a new member P is created and added to the LIST F.

Likewise

```
      FOR F, Q WITH A EQ '500'
           P = P;
        END;
```

has the same effect as

```
      F = Q WITH A EQ '500'
```

A final example: with FF and PP as given in the example for the matching rules,

```
     FOR PP, FF WITH STATE EQ 'RI'
                 OR STATE EQ 'CT' OR STATE EQ 'MA'
                 OR STATE EQ 'VT' OR STATE EQ 'NH'
                 OR STATE EQ 'ME'
           NAME = NAME;
     END;
```

will have the effect of outputting through the PORT PP, the NAMEs of all PERSONs in the FILE FF who live in New England; i.e. with STATE equal to one of the New England states.

<u>UPDATE</u>

With UPDATE a user can replace the contents of any container which is neither variable length nor inverted. Given a FILE and PORT:

```
     CREATE FAMILIES FILE LIST
       FAMILY STRUCT
        MOTHER STR (,8), C=1
        FATHER STR (,8), C=1
        KIDS LIST (,10), C=1
          KID STRUCT
            NAME STR (,6), C=1
            AGE STR (2)
            KID%NUM BYTE, V=I
          END
        FAM%NUM BYTE, V=I
      END;

      CREATE FAMILIES PORT LIST
         FAMILY STRUCT
           MOTHER STR (,8), P=EOR
           FATHER STR (,8), P=EOR
```

```
          KIDS LIST (,10), P=EOB
            KID STRUCT
              NAME STR (,8), P=EOR
              AGE STR (2), P=EOR
            END
        END;
```

then, with the FILE FAMILIES open in the WRITE MODE:

```
          UPDATE FAMILIES WITH
          MOTHER EQ MOTHER AND FATHER EQ FATHER,
          PAMILIES
          UPDATE KIDS WITH NAME EQ NAME, KIDS
          AGE=AGE; END; END;
```

When UPDATing, the datacomputer finds the first match on the outer LIST (FAMILIES); then, if one is requested, the first match on an inner LIST (KIDS). The datacomputer does the UPDATE and proceeds to the next match. The above UPDATE will match the transaction PORT on MOTHER, FATHER (in the outer LIST), then on the NAME (in the inner LIST) and replace the AGE with the AGE from the transaction PORT. Only the first match is UPDATEd; if others exist, they will not be found. The EQ specifies those elements which must match exactly between the master FILE and the transaction PORT; the "=" is the UPDATing assignment of the transaction PORT element which will replace the master FILE element. Since the datacomputer is making a sequential pass of PORT and FILE, the information appearing in the transaction PORT must occur in the same order as it appears in the master FILE; that is, outer and inner PORT LIST members must be in the same order as those of the FILE. LIST members which contain no information different from the master FILE (which are not being changed by the UPDATE) may be omitted.

If the datacomputer fails to find a match on MOTHER, FATHER or NAME, it will give the error message (see Chapter 6):

```
;U000 dd-mm-yy hhmm:ss LEBARF: NO MATCH FOUND
```

Any UPDATing information occurring after the failure to match will be discarded. This happens because the FILE being UPDATEd is searched to its end for a match and the datacomputer presently has no method of searching more than once through a portion of the master FILE.

It is possible to do an UPDATE with qualifications both on the master and transaction LISTs (...FAMILIES WITH MOTHER EQ 'ANITA', PAMILIES WITH FATHER EQ 'JOHN'...). In such a case, the datacomputer finds the match on the transaction LIST and then the match on the master LIST.

A common process would be to change some existing data within a FILE and then to APPEND further information to the end of a FILE. Care must be taken when doing this, to remember to change the FILE MODE. An UPDATE requires the FILE be in WRITE MODE, but if the MODE is not changed to APPEND before attempting to add further data, what has already been written in the FILE will be replaced.

Although an inverted container may not be changed by an UPDATE in Version 0/11, if matching for the UPDATE refers to an inverted container, the datacomputer makes use of the inversion to perform the UPDATE more efficiently. The requirement of exactly similar ordering still remains.

An UPDATE may be done without a transaction PORT or FILE. For example:

```
UPDATE FAMILIES WITH MOTHER EQ 'ANITA'
UPDATE KIDS WITH NAME EQ 'JULIE'
AGE='24' END; END;
```

will determine that LIST-member with specified MOTHER and NAME and change the associated AGE to 24. When doing an UPDATE such as this one, in which the information is contained in the request, the datacomputer will give no message if the requested matches on MOTHER and NAME are not found. Also, this type of UPDATE will change all instances that fulfill the specifications, not just the first one.

UPDATing is one method by which a number of elements can be changed with a single "= quoted constant" assignment. If the restriction WITH NAME EQ 'JULIE' is dropped from the above UPDATE all KIDS in FAMILIES WITH MOTHER EQ 'ANITA' will be given an AGE equal to '24'.

<u>Mismatched FOR Loops</u>

It is possible, by using a mismatched FOR-loop, to output selected information from the FILE LIST.

```
CREATE MISFAM PORT LIST
  KID STRUCT
   FATHER STR (,8), P=EOR
   NAME STR (,8), P=EOR
   AGE STR (2), P=EOR
   END;
```

then

```
FOR FAMILIES
FOR MISFAM, KIDS
FATHER=FATHER; NAME=NAME; AGE=AGE;
END; END;
```

will output all children's names and ages and the father's name for each child. The selection may be further narrowed by reference to the virtual index containers.

```
FOR FAMILIES WITH FAM%NUM GT 5
FOR MISFAM, KIDS WITH KID%NUM GE 1
FATHER=FATHER; NAME=NAME; AGE=AGE;
END; END;
```

This will select the FATHER, KID NAME and AGE of those FAMILIES with one or more child, which are not in the first five FAMILIES.

Chapter 6: Using the Datacomputer

We proceed now from the basics of the language itself, such as containers and assignment, to a broader view of how datalanguage might be employed by a user's program. We will discuss such matters as accessing the datacomputer, transmitting data to and from datalanguage PORTs, and various aids to the maintenance of data and FILE and PORT descriptions on the datacomputer.

Interacting with the Datacomputer

Typically, datalanguage requests will be sent to the datacomputer by a user program residing on some computer on the ARPA network. All interaction between the user program and the datacomputer takes place over the network.

Information transmission over the network takes place along uni-directional paths. For a two-way conversation, two such paths are needed, one for transmission in each direction. The end of a transmission path is called a socket; a socket can be either a send (output) or receive (input) socket. Obviously, a transmission path requires a send socket at one end and a receive socket at the other. A diagram of the sockets involved in a two-way conversation over the network appears below.

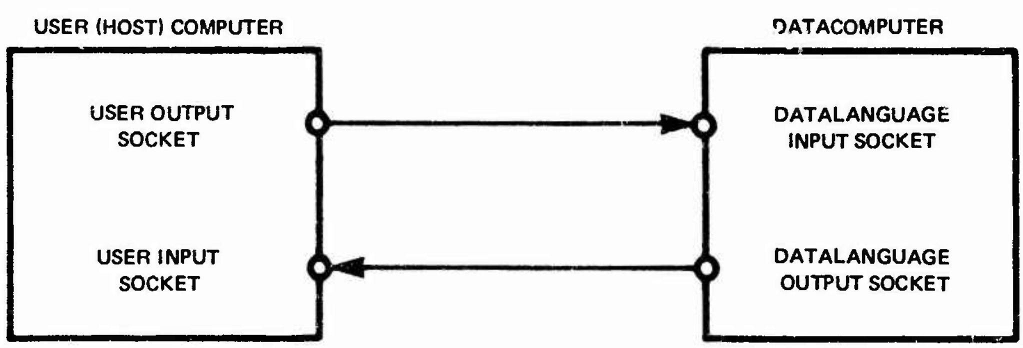

Figure 4-1. Network connections to the datacomputer

A host computer is identified on the network either by a number or by an alphabetic name, like BBN-TENEX. A socket within a given host is identified by a number; send sockets

have odd numbers and receive sockets even ones. For a connection to be opened, both hosts involved must request that it be opened. Likewise, after data transmission is complete, both hosts must close their ends of the connection. The period of time during which network connections are open between a user host and the datacomputer is called a session.

In the user program's dialogue with the datacomputer, the transmission in one direction consists largely of datalanguage requests, while messages from the datacomputer are sent in the other direction, to the user program. The sockets at the datacomputer that are used for these purposes are called the datalanguage input socket and the datalanguage output socket. The terms datalanguage input/output port are also used. These ports, like the PORTs that a user can create with datalanguage CREATE requests, are channels for the input and output of information. However, the purpose of the datalanguage ports is to receive datalanguage and transmit datacomputer messages; the purpose of a user PORT is to transmit or receive data.

The protocol by which a user program can set up datalanguage input and output sockets connected to its own output and input sockets is described in Appendix D of this document.

Synchronization

Since use of the datacomputer typically involves the interaction of two programs at opposite ends of a communication network with a finite time delay, steps must be taken to ensure that the programs remain in synchrony with each other. If they do not, the user program might blithely go on sending datalanguage when the datacomputer expects data or might receive diagnostic messages when it expects a list of directory node names.

To avoid such problems, the datacomputer generates a variety of messages that keep the user program informed of what is going on. The messages fall into several categories: there are error messages, which will be discussed in a later section; informational messages, which can safely be ignored or merely logged by a user program; and synchronization messages, some of which at least must be processed by the user program to ensure proper communication. The first character of the message differs from category to category, allowing the user program easily to differentiate the various classes of message.

Prefix	Type of Message
?, -, or +	error message
;	informational message
.	synchronization message

Other special characters may be added as datacomputer message-prefix characters in future versions. The letters, digits, tab, and space will never be used as message prefixes, however.

The datacomputer's messages all follow a common format, which includes the special header character just described, a letter and three digits that a program can use to identify the message, the date and time of the message's transmission, and a variable-length string of text that can be read by a human user. Specifically, the format is:

```
.X999 dd-mm-yy hhmm:ss (TAB) TEXT STRING (CR, LF)
```

where . represents the header character, X999 represents the message identifier (for example, I210), dd-mm-yy represents the day, month, and year (for example, 25-09-73), hhmm:ss represents the time on a 24-hour clock in hours, minutes, and seconds, (TAB) represents a tab character, and (CR, LF) represents the carriage return, line feed characters that terminate the message. All alphabetic characters in the message are capitalized. Note that the message may be very long (too long to print on a 72-column printer, for instance), so a user program that processes datacomputer messages may have to format them to be readable.

In this manual, only the invariant parts of messages will be displayed; that is, the header character, the identifying letter and digits, and the message text.

To illustrate the use of synchronization messages in pacing interaction with the datacomputer, consider these two:

```
.I210 LAGC: READING NEW DL BUFFER
.J900 FCFINI: END OF SESSION
```

The first message, .I210, is sent by the datacomputer over the datalanguage output socket, and hopefully received by the user program over an input socket, whenever the datacomputer is ready to accept datalanguage requests. The user program will in general respond to this message by transmitting a <u>line</u> of datalanguage. A line is some number of characters (currently there is an upper limit of about 2500) terminated by either the character sequence carriage

return, line feed (ASCII codes 15, 12 octal) or the single character eol (37 octal). On a line may be one datalanguage request (terminated by a semicolon), several requests (each terminated by a semicolon), or a portion of a request.

In the first two cases, when the datacomputer receives the requests (and if they contain no errors) it will proceed to execute them, (typically generating messages and/or initiating data transfers as it does). Following execution, it will again send the .I210 message signifying that it is again ready to receive datalanguage. In the third case, the datacomputer will continue to send .I210 messages, prompting the user program for lines of datalanguage, until a complete request has been assembled; the request will then be executed as described above.

The second message, .J900, is sent by the datacomputer at the end of a session. The user program may request that the session end by sending the datacomputer a control-Z (ASCII code 32 octal) in response to a .I210 message. The datacomputer responds to control-Z by executing an end of session procedure, which involves closing any open containers, deleting TEMP PORTs, and sending the .J900 message. The user program may then close its network connections with the datacomputer.

Synchronization after an error is discussed in the section entitled Error Messages below.

Transmitting Data through the Datalanguage Ports

Often, a user program will need to send data over the network to be stored at the datacomputer, or to process data that it receives from the datacomputer. If all of the data is described as ASCII, then this may be done by using the datalanguage input or output port.

To reference data that will be transmitted through the datalanguage input socket, the user need only open a PORT and use it on the right-hand side of an assignment in datalanguage. When the assignment is executed, data will be accepted through the datalanguage input port and assigned to whatever container appears on the left side of the request.

Similarly, to output data through the datalanguage output socket so that it can be picked up by the user program, all that is needed in datalanguage is a PORT used on the left-hand side of an assignment. Any data assigned to that container will be transmitted through the datalanguage output port over the network.

Of course, this requires the use of more synchronization messages. To treat the data-input case first:

.I231 OCPBO: (DEFAULT) INPUT PORT OPENED
.I251 OCPBC: (DEFAULT) INPUT PORT CLOSED

After the user program has sent the datalanguage assignment request that references the open input PORT, the datacomputer will transmit the .I231 message over the datalanguage output port. The message signals that input data is now expected through the datalanguage input port, and the user program should send the data. Data transmission is terminated by a control-Z character which causes the datacomputer to send the .I251 message confirming that data transmission is finished. The next synchronization message will be .I210, a request for more datalanguage.

The synchronization procedure governing data output through the datalanguage output port is similar. The messages are

.I241 OCSOP: (DEFAULT) OUTPUT PORT OPENED
.I261 OCSCL: (DEFAULT) OUTPUT PORT CLOSED

When the assignment statement is executed which requests that data be output through the datalanguage port, the datacomputer first sends .I241, followed by the requested data, followed in turn by .I261. The datacomputer does not output a control-Z at the end of the data. The user program can use these messages to separate the data from all other information.

Opening a Secondary Port

Instead of a datalanguage port, an additional network connection or secondary port can be used for transmitting data. Non-ASCII data, including an ASCII STR with a preceding count or a non-ASCII delimiter, must be transmitted over a secondary port (see Chapter 2, delimiter). The CONNECT request sets up the secondary port.

The CONNECT request names an open PORT, and gives a host (that is, a computer on the network) and socket number to which that PORT is to refer. As mentioned above, if a CONNECT request is never executed for a PORT, it will refer to the socket from which the user program transmits datalanguage (if it is a READ PORT) or the socket at which the user program receives the datacomputer's messages (a WRITE or APPEND PORT). The form of the CONNECT request is

CONNECT <pathname> <address> ;

where <pathname> is the node name, complete name (i.e. starting with $TOP) or simple login name (i.e. starting immediately subordinate to the login node) of an open PORT, and <address> can have several forms. It can be one of

<socket-no> the decimal number of a socket at the user's host computer.

<host-no> <socket-no> where <host-no> is the decimal number of a computer on the ARPA network

'<host-name>' <socket-no> where <host-name> is the host computer's TENEX alphabetic name

<host-name> <socket-no> where <host-name> is the host computer's TENEX alphabetic name (such as 'CCA')

OR '<local-file-designator>' This last form of <address> does not refer to the network, but is included here for completeness. <local-file-designator> is a TENEX file designator that refers to a file at the datacomputer site.

A CONNECT may be executed any time the PORT is open, but it does not actually establish the network connection. Those connections are established, used, and then closed again during the execution of an assignment statement in datalanguage, and CONNECT merely sets up the socket address to be used when the PORT is later referenced in an assignment.

A DISCONNECT request may be used to cause a CONNECTed PORT to refer once again to the datalanguage input or output port.

DISCONNECT <pathname> ;

Two CONNECT requests may be issued for the same PORT without an intervening DISCONNECT.

Additional synchronization messages are generated at the time a CONNECTed PORT is used in an assignment statement. These messages are

```
.I230 OCPBO: OPENING INPUT PORT
;I239 OCPBO: INPUT PORT OPENED
.I250 OCPBC: CLOSING INPUT SOCKET
;I259 OCPBC: INPUT SOCKET CLOSED
```

.I240 OCPOO: OPENING OUTPUT PORT
;I249 OCPOO: OUTPUT PORT OPENED
.I260 OCPOC: CLOSING OUTPUT SOCKET
;I269 OCPOC: OUTPUT SOCKET CLOSED

When a CONNECTed PORT is used on the right-hand side of an assignment (that is, in READ mode), the .I230 message is sent over the datalanguage output port. This signals the user program that the datacomputer is attempting to open a network connection to the host and socket specified by the CONNECT request for the PORT. The user program should thus open its end of the connection itself (if it is a connection to a different socket on the user program's own host) or ensure that the third host opens its end of the connection at this time (if it is a connection to another host on the network).

The ;I239 message indicates that indeed the network connection was opened correctly. After this message is received, data can be transmitted, terminated by closing the network connection. When the connection is being closed, the datacomputer sends .I250 over the datalanguage output port and then ;I259, signaling the user program that use of the secondary network connection is complete. The .I250 may precede or follow the closing of the connection on the user's side.

The messages for output PORTs work similarly, with .I240 signaling that the output network connection is being opened, ;I249 that the connection is opened, and .I260 that output is complete and the connection is being closed, and ;I269, that the closing has been completed.

If there are errors in the data, other messages will be sent before the .I250 or .I260 message. This would be the case, for example, if the data does not match the description (see Appendix G).

A user program can interrupt the datacomputer's transmission of data; see Appendix D for details.

The form CONNECT <pathname> <local-file-designator>; may be useful to those with large amounts of data to send to the datacomputer. In some cases, the shipment of magnetic tapes by air-freight produces higher bit rates than sending the data over the network; the magnetic tape may then be addressed from datalanguage as a local file. Contact CCA for information on this procedure.

Error Messages

Datacomputer error messages will in general be seen by a human user, although they have header characters which make them potentially processable by a smart user program. Error messages fall into several categories, distinguished by their first character.

First Character	Meaning
?	indicates a datacomputer or system bug. A user program should rarely see one of these.

Examples:

```
?U000 TRDN: NODE CHAIN SNAFU
?U000 DKWR: DISK I/O WRITE ERROR
```

-	indicates a user error -- typically bad datalanguage, data, or i/o handling. A debugged user program should rarely see one of these. (See Appendix G)

Examples:

```
-U000 LPNM: FORARG NOT DIRECT LIST MEMBER
-I246 OCSOP: CAN'T OPEN OUTPUT PORT (BAD CONNECT ARGS?)
```

+	indicates a circumstantial error, such as a file's being busy, or an error which is due to current datacomputer limitations.

Examples:

```
+U000 OCDOP: CAN'T OPEN FILE (SOMEBODY ELSE UPDATING?)
+L000 DHIN: DESCRIPTOR TOO LARGE
```

After the datacomputer generates one or more error messages, it follows a special procedure to resynchronize itself with the user. This procedure involves waiting for a special character, control-L or form feed (ASCII 14 octal), to be transmitted by the user. That is, after the error message the datacomputer sends

```
.I220 LAEB: LOOKING FOR CONTROL-L
```

This is repeated for each line of input it receives on the datalanguage input port until the user sends a control-L character. Following receipt of a control-L, .I210 will

again be sent and datalanguage requests again processed.

More severe action must be taken following certain system or ?-type errors. One of the following synchronization messages may be generated:

.J151 FCERRH: RESTARTING THE REQUEST HANDLER
.J140 FCREIN: REINITIALIZING USER JOB
.J910 FCERRH: CRASHING JOB

The .J151 message indicates that TEMP PORTs have been deleted; otherwise, the status of the session remains the same (PORTs and FILEs will still be open, etc.). This message will usually be followed by .I220, a request for control-L.

The .J140 message is more serious. The user's job is completely reinitialized, leaving his status the same as when the session was begun. This message will also be followed by .I220.

The .J910 message indicates a condition so severe that the datacomputer does not know how to recover. The user's job is crashed and the datalanguage network connections closed. That is, the session is forcibly ended.

If this happens, and also if the user's network connections to the datacomputer are accidentally broken, the datacomputer will do its best to close his open PORTs and FILEs in an orderly manner. However, if the user was in the process of transmitting data into a FILE, the last few thousand characters of data his program sent may have been lost in transit and not incorporated into the FILE.

Not much in general can be said about handling ? or - errors, except that a human user will have to read and interpret the text of the error message in each case, and (in the case of - errors) correct the datalanguage he is having his program send.

Appendix G of this manual is a listing of error messages commonly caused by bad datalanguage and errors in data streams, with a few examples of the type of datalanguage that could cause these messages.

+ errors, on the other hand, could be processed by a user program. The most reasonable thing to do in many cases is to wait five minutes and retry the datalanguage request that caused the error. For example, a FILE which was busy (i.e. in use by someone else) may be free by that time, so the second attempt to use it may be successful.

Messages beginning with +L are an exception to this, in that the appropriate time to wait may be several weeks instead of minutes. Such messages indicate limitations of the current datacomputer system, such as limitations imposed

by internal table sizes. A new version of the datacomputer may remove many of these limitations. Realistically, this means that +L messages are like - messages in that a program probably could not handle them.

Addendix A: Syntax for 0/11

The following is the complete BNF (Backus Normal Form) specification of datalanguage syntax for version 0/11 of the datacomputer.

Requests

```
<request>  ::=  ;
```

Directory Requests

```
<request>  ::=  LOGIN <login body> ;
     |  CREATE <create body> ;
     |  DELETE <delete body> ;
     |  OPEN <open body> ;
     |  CLOSE <close body> ;
     |  CONNECT <connect body> ;
     |  DISCONNECT <disconnect body> ;
     |  MODE <mode body> ;
     |  CREATEP <createp body> ;
     |  DELETEP <deletep body> ;
     |  LIST <list body> ;
```

Data Transfer Requests

```
<request>  ::=  <direct assignment> ;
     |  <loop> ;
```

Directory

Pathnames

```
<pathname>  ::=  <complete pathname>
    |  <simple complete pathname>
    |  <login pathname>
    |  <simple login pathname>
    |  <open node name>

<node name>  ::=  <identifier>
    |  <identifier> ( <password string> )
<password string>  ::=  <string constant>
<simple node name>  ::=  <identifier>

<complete pathname>  ::=  %TOP . <node name>
    |  <complete pathname> . <node name>

<simple complete pathname>  ::=
    %TOP . <simple node name>
    |  <simple complete pathname> . <simple node name>

<login pathname>  ::=  <node name>
    |  <login pathname> . <node name>

<simple login pathname>  ::=  <simple node name>
    |  <simple login pathname> . <simple node name>

<open node name>  ::=  <simple node name>

<node pathname>  ::=  <complete pathname>
    |  <login pathname>

<open pathname>  ::=  <simple complete pathname>
    |  <simple login pathname>
    |  <open node name>
```

Directory

Requests

```
<login body>  ::=  %TOP
   |  <node pathname>

<create body>  ::=  <simple node name>
   |  <node pathname> . <simple node name>
<create body>  ::=  <data description>
   |  <node pathname> . <data description>

<delete body>  ::=  **
   |  <login pathname>
   |  <login pathname> . **

<open body>  ::=  <node pathname>
   |  <node pathname> <mode>

<close body>  ::=  %OPEN
   |  <open pathname>

<connect body>  ::=
   <open pathname> <tenex file specification>
   |  <open pathname> <network specification>
<tenex file specification>  ::=  <string constant>
<network specification>  ::=  <socket number>
   |  <host specification> <socket number>
<socket number>  ::=  <integer constant>
<host specification>  ::=  <integer constant>
   |  <identifier>
   |  <string constant>

<disconnect body>  ::=  <open pathname>

<mode body>  ::=  <open pathname> <mode>
<mode>  ::=  READ
   |  WRITE
   |  APPEND
   |  WRITE DEFER
   |  APPEND DEFER
```

```
<createp body>  ::=  <node pathname>
   ¦  <node pathname> <privilege tuple specification>
<privilege tuple specification>  ::=
   <privilege tuple option>
   ¦  <privilege tuple specification>
           <privilege tuple option>
<privilege tuple option>  ::=  , U = <user identity>
   ¦  , H = <host identity>
   ¦  , S = <socket identity>
   ¦  , P = <password string>
   ¦  , G = <grant privilege list>
   ¦  , D = <deny privilege list>
   ¦  , N = <privilege tuple number>
<user identity>  ::=  **
   ¦  <user node>
   ¦  <user node set>
   ¦  <user node> . **
   ¦  <user node set> . **
   ¦  <user node> . <user node set> . **
<user node>  ::=  <identifier>
   ¦  <user node> . <identifier>
<user node set>  ::=  *
   ¦  <user node set> . *
<host identity>  ::=  ANY
   ¦  LOCAL
   ¦  <integer constant>
<socket identity>  ::=  ANY
   ¦  <integer constant>
<grant privilege list>  ::=  <grant privilege>
   ¦  <grant privilege list><grant privilege>
<grant privilege>  ::=  C
   ¦  L
   ¦  R
   ¦  W
   ¦  A
<deny privilege list>  ::=  <deny privilege>
   ¦  <deny privilege list><deny privilege>
<deny privilege>  ::=  R
   ¦  W
   ¦  A
<privilege tuple number>  ::=  <integer constant>

<deletep body>  ::=
   <node pathname> <privilege tuple number>
```

```
<list body>  ::=  <list node set>
    |  <list node set> <list option>
<list node set>  ::=  %TOP
    |  %OPEN
    |  *
    |  **
    |  <open node name>
    |  <node pathname>
    |  <node pathname> . *
    |  <node pathname> . **
<list option>  ::=  %NAME
    |  %DESCRIPTION
    |  %DESC
    |  %SOURCE
    |  %ALLOCATION
    |  %ALLOC
    |  %PRIVILEGE
    |  %PRIV
```

Data Description

```
<datatype>  ::=  <compound datatype>
    |  <simple datatype>
    |  <string>

<compound datatype>  ::=  LIST
    |  <structure>
<structure>  ::=  STRUCTURE
    |  STRUCT

<simple datatype>  ::=  BYTE
    |  <integer>
<integer>  ::=  INTEGER
    |  INT

<string>  ::=  <string type>
    |  <string type> <string interpretation>
<string type>  ::=  STRING
    |  STR
<string interpretation>  ::=  ASCII
    |  ASCII8
    |  BYTE
    |  INTEGER
    |  INT
```

```
<data description> ::=
   <simple node name> <function>
           <outermost description>
<function> ::= FILE
   ¦ PORT
   ¦ TEMPORARY PORT
   ¦ TEMP PORT
<outermost description> ::= LIST <description>
   ¦ LIST <list options> <description>
   ¦ <string>
   ¦ <string> <string options>
   ¦ <description body>

<descriptions> ::= <description>
   ¦ <descriptions> <description>
<description> ::=
   <description name> <description body>

<description name> ::= <identifier>
<description body> ::=
   LIST <dimension> <description>
   ¦ LIST <dimension> <list options> <description>
   ¦ <structure> <descriptions> END
   ¦ <structure> <compound datatype options>
          <descriptions> END
   ¦ BYTE
   ¦ BYTE <byte options>
   ¦ <integer>
   ¦ <integer> <simple datatype options>
   ¦ <string> <dimension>
   ¦ <string> <dimension> <string options>
```

```
<description option> ::= <inversion option>
    | <byte size option>
    | <filler option>
    | <variable length option>
    | <virtual data option>
    | <container address table option>
<inversion option> ::= , I = D
    | , I = I
<byte size option> ::= , B = <integer constant>
<filler option> ::= , F = <integer constant>
    | , F = '<nonquote character>'
<variable length option> ::= , C = 1
    | , P = EOF
    | , P = EOB
    | , P = EOR
    | , D = <integer constant>
    | , D = '<nonquote character>'
<virtual data option> ::= , V = I
<container address table option> ::= , CAT

<compound datatype options> ::=
   <compound datatype option>
    | <compound datatype options>
          <compound datatype option>
<compound datatype option> ::= <byte size option>
    | <filler option>
    | <variable length option>

<simple datatype options> ::=
   <simple datatype option>
    | <simple datatype options>
          <simple datatype option>
<simple datatype option> ::= <inversion option>
    | <byte size option>
    | <filler option>

<string options> ::= <string option>
    | <string options> <string option>
<string option> ::= <inversion option>
    | <byte size option>
    | <filler option>
    | <variable length option>

<list options> ::= <list option>
    | <list options> <list option>
<list option> ::= <compound datatype option>
    | <container address table option>
```

```
<byte options>  ::=  <byte option>
    |  <byte options> <byte option>
<byte option>  ::=  <simple datatype option>
    |  <virtual data option>

<dimension>  ::=  ( <integer constant> )
    |  ( , <integer constant> )
    |  ( <integer constant> , <integer constant> )
```

Data Transfer

```
<data reference>  ::=  <identifier>
   |  <data reference> . <identifier>
<constant>  ::=  <string constant>
   |  <integer constant>
<assignment>  ::=  <data reference> = <data reference>
   |  <data reference> = <constant>

<direct assignment>  ::=  <assignment>
   |  <implicit for loop>
<implicit for loop>  ::=  <assignment> <qualifier>

<loop>  ::=  <for loop>
   |  <update loop>

<for loop>  ::=  FOR <input> <loop body> END
   |  FOR <input> <qualifier> <loop body> END
   |  FOR <output> , <input> <loop body> END
   |  FOR <output> , <input> <qualifier>
           <loop body> END
<input>  ::=  <data reference>
<output>  ::=  <data reference>

<update loop>  ::=  UPDATE <master> <loop body> END
   |  UPDATE <master> <qualifier> <loop body> END
   |  UPDATE <master> , <transaction> <loop body> END
   |  UPDATE <master> <qualifier> , <transaction>
           <loop body> END
<master>  ::=  <data reference>
<transaction>  ::=  <data reference>

<loop body>  ::=  <loop>
   |  <loop> ;
   |  <assignment list>
   |  <assignment list> ;
<assignment list>  ::=  <assignment>
   |  <assignment list> ; <assignment>

<qualifier>  ::=  WITH <boolean expression>

<boolean expression>  ::=  <relational expression>
   |  ( <boolean expression> )
   |  NOT <boolean expression>
   |  ANY <boolean expression>
   |  <boolean expression> AND <boolean expression>
   |  <boolean expression> OR <boolean expression>

<relational expression>  ::=
   <data reference> <comparison operator>
           <data reference>
   |  <data reference> <comparison operator> <constant>
```

```
<comparison operator>  ::=  EQ
    |  NE
    |  GT
    |  GE
    |  LT
    |  LE
```

Lexical Items

```
<lexical item>  ::=  <identifier>
    |  <integer constant>
    |  <string constant>
    |  <autonomous character>

<identifier>  ::=  <letter>
    |  $
    |  <identifier> <letter>
    |  <identifier> $
    |  <identifier> <digit>

<integer constant>  ::=  <digit>
    |  <integer constant> <digit>

<string constant>  ::=  '<string constant body>'
<string constant body>  ::=  <nonquote character>
    |  <string constant body> <nonquote character>
```

Character Set

```
     <letter>  ::=  A
        ¦  B
                ...
        ¦  Z
        ¦  a
        ¦  b
                ...
        ¦  z

     <digit>  ::=  0
        ¦  1
                ...
        ¦  9

     <nonquote character>  ::=  <letter>
        ¦  %
        ¦  <digit>
        ¦  <autonomous character>
        ¦      (space)
        ¦      (horizontal tab -- HT)
        ¦  "'
        ¦  ""

     <separator>  ::=      (space)
        ¦      (horizontal tab -- HT)
        ¦  <eol>
     <eol>  ::=      (end of line -- octal 37)
        ¦  <carriage return> <line feed>
     <carriage return>  ::=      (carriage return -- CR)
     <line feed>  ::=      (line feed -- LF)
```

```
<autonomous character>  ::=  !
                          |  #
                          |  $
                          |  &
                          |  (
                          |  )
                          |  *
                          |  +
                          |  ,
                          |  -
                          |  .
                          |  /
                          |  :
                          |  ;
                          |  <
                          |  =
                          |  >
                          |  ?
                          |  @
                          |  [
                          |  \
                          |  ]
                          |  (up arrow)
                          |  (left arrow)
                          |  (at sign)
                          |  {
                          |  (vertical bar)
                          |  }
                          |  (up arrow)
```

Notes

Character codes are 7 bit ASCII.

Separators are always permitted between lexical items, except between grant privileges, between deny privileges, and inside string constants.

Comments may be inserted wherever separators are allowed. Comments begin with '/*' and end with '*/' (e.g., /* THIS IS A COMMENT */).

<carriage return> and <line feed> may appear together only in that order (as an <eol>). Otherwise they are treated as control characters, which are rejected.

Appendix B: Reserved Words

AND
ANY
ASCII
ASCII8
BYTE
CLOSE
CONNECT
CREATE
CREATEP
DELETE
DELETEP
DISCONNECT
END
EQ
FILE
FOR
GE
GT
INT
INTEGER
LE
LIST
LOGIN
LT
MODE
NE
NOT
OPEN
OR
PORT
STR
STRING
STRUCT
STRUCTURE
UPDATE
WITH
%OPEN
%TOP

Appendix C: Inversion: Technical Considerations

An inversion is a secondary data structure that the datacomputer can use to improve its efficiency in retrieving data by content from a FILE. Specifically, an entry in the inversion is constructed for every container with the inversion attribute. For each data value which occurs for the container, the inversion contains pointers to all the records in the FILE for which that container has that value.

For example, if

```
CREATE PEOPLE FILE LIST
     PERSON STRUCT
        NAME STR (15)
        SOCSECNO STR(9),I=D
        SEX STR (1) /* 'M' OR 'F'*/,I=D
        ZIP STR(5),I=D
     END;
```

then the data structure for the inversion on SEX contains pointers to all instances of PERSONs with SEX equal to 'F', and similarly for 'M'. Thus, evaluation of a FOR input-spec like

```
FOR ... , PEOPLE WITH SEX EQ 'M'
```

would not require a full sequential reading of the FILE PEOPLE.

An inversion is not only constructed automatically by the datacomputer when the FILE is loaded with data, but is automatically maintained whenever information is appended to the FILE.

Unfortunately, even if an inversion for the appropriate container exists, the datacomputer cannot always use it for the evaluation of input-specs, and must sometimes resort to time-consuming searches of the FILE. In particular, the inversion can be used only when the container is compared with a constant using the operators EQ and NE. That is,

```
PERSON WITH ZIP EQ '02138' OR ZIP EQ '02139'
      OR ZIP EQ '02140' OR ZIP EQ '02141'
```

can be evaluated directly from the inversion. However,

```
PEOPLE WITH ZIP GE '02138'
                  AND ZIP LE '02141'
```

while it still can be evaluated, cannot take advantage of the inversion and so would normally be much less efficient.

Furthermore, when the container is a member of an inner LIST, only the operator EQ can be evaluated using the inversion. A sequential search is used for evaluating NE.

Complex Boolean expressions, those involving several comparisons, fall into three classes: those with all comparisons evaluable from the inversion, those containing no comparisons evaluable from the inversion, and those which mix the two kinds of comparisons. The first two classes pose no problem; the datacomputer will use the inversion to evaluate expressions in the first category, and not for expressions in the second category.

For mixed expressions, the datacomputer will use the inversion as much as it can. For the present, this can be stated as follows: if the Boolean expression is of the form

```
<expr> AND <expr> AND ...
```

(where <expr> is an arbitrary Boolean expression, in parentheses if it contains OR, then the datacomputer will separate the <expr>s into those that can be completely evaluated from the inversion and those that cannot, and will process those for which it can use the inversion first. The <expr>s that cannot use the inversion are evaluated by an exhaustive search of the set of records selected by the earlier <expr>s.

For an example, take the above FILE, PEOPLE. Suppose a list of all males with ZIP GT '02000' were desired. ZIP is indeed inverted, but since the operator GT is involved, the evaluation of that part of the Boolean expression cannot use the inversion. As a result, in

```
FOR ... , PEOPLE WITH  ZIP GT '02000' AND SEX EQ 'M'
```

the datacomputer will first use the inversion to find the set of all PERSONs with SEX EQ 'M', and only this smaller set of PERSONs would be searched for the desired ZIPs.

A more difficult example: consider the problem of retrieving all the records for events that occurred between 10:05 on the 25th and 15:07 of the 30th from a FILE that is inverted on DAY but not on TIME. A straightforward way to do this is

```
... WITH (DAY EQ '25' AND TIME GT '10:05')
        OR (DAY EQ '26') OR (DAY EQ '27') OR ...
         OR (DAY EQ '30' AND TIME LT '15:07')
```

but this is quite inefficient: the inversion cannot be used at all, for this Boolean expression is mixed and is not set up as a series of terms connected by AND. The best way to express this condition is

```
    ... WITH (DAY EQ '25' OR DAY EQ '26' OR ... OR DAY EQ
'30')
            AND (DAY NE '25' OR TIME GT '10:05')
               AND (DAY NE '30' OR TIME LT '15:07')
```

In this case, only records for the correct six days are retrieved by the first term, so only they need to be searched through for the evaluation of the second and third terms.

Future versions of the datacomputer will automatically optimize mixed Boolean expressions, freeing the user from this task.

The computation of the space requirements for an inversion is best left to the datacomputer's operational staff at CCA, who should be contacted by any user interested in setting up a data file with an inversion.

Appendix D: Network Interaction with the Datacomputer

The procedure for establishing network connections with the datacomputer is that documented in J. Postel, Official Telnet - Logger Initial Connection Protocol, NIC 7103, 15 June 1971. The following is a simplified, informal description of that procedure.

The datacomputer listens for connections on a well-advertised socket, currently number 103 (octal) at CCA, host number 37 (octal). This is an odd-numbered or send socket. The user program wishing to use the datacomputer will address this socket from a socket on his own host computer -- say from socket number U. U must, of course, be an even number or a receive socket. The user program should read one 32-bit byte of information over this connection and then immediately close it (leaving socket CCA-103 free for other users). This byte of information is a socket number at the datacomputer -- say socket D. D will be an even number.

The last step is the opening of two network connections, the permanent datalanguage connections. They are

from D+1 at CCA to U+2 at the user host
and from U+3 at the user host to D at CCA.

Note that U+2 is even (since U is) and D+1 is odd -- this is the datalanguage output socket. Also, U+3 is odd, and D is even: the datalanguage input socket. These connections will remain in effect until the end of the datalanguage session.

Two special network control signals, INS and INR, may be used to interrupt the datacomputer. INS, for interrupt the sender, may be sent at any time during the processing of a request and stops data output from the current request. No error message or other acknowledgement will be generated; the output simply stops. INS might be useful to a program which receives output from the datacomputer and displays it to a human operator sitting at a teletype; at the request of the user, the program could send INS to stop an overly-long printout.

INR, for interrupt the receiver, performs all the functions of INS. In addition, compilation or any other processing that is under way when INR is received will be aborted, possibly generating an error message and a request

for control-L. INR thus requests a more immediate halt than does INS.

Appendix E: Implementation Restrictions

A number of datalanguage restrictions specific to Version 0/11 are collected here for ready reference. Note that some of these restrictions have been mentioned in the body of this manual, while others have not.

1. There is a restriction on the containers that can be referenced in the body of a FOR-loop. Consider the following example:

```
   CREATE FF FILE LIST
       PERSON STRUCT
           NAME STR (15)
           ADDRESS STR (20)
           CITY STR (10)
           STATE STR (2)
           ZIP STR (5)
           SOCSECNO STR (9)
           DEPENDENTS LIST (10)
               NAME STR (15)
       END;
   CREATE PP PORT LIST
       PERSON STRUCT
           NAME STR (15)
           SOCSECNO STR (9)
       END;
```

To output all the DEPENDENTS.NAMEs from the file FF, together with the SOCSECNO of the PERSON whose DEPENDENTS they were,

```
   FOR PP,FF
       NAME=NAME;
       SOCSECNO=SOCSECNO;
   END;
```

This example as written will work in datalanguage 0/11. However, if SOCSECNO occurred after DEPENDENTS in the description of PERSON in FF, the request would fail due to a compiler restriction.

When an inner FOR-loop is processing a LIST which occurs within a STRUCT, references may be made in the body of that FOR to objects which occur before that LIST in the STRUCT, but not after the LIST.

There are certain cases of assignment involving inner LISTs which the compiler in Version 0/11 cannot

handle. For example, given two structures of the following format:

```
L1 FILE LIST
     S1 STRUCT
          A1 STR (8)
          A2 LIST (4)
               B2 STR (6)
     END;
```

and

```
L2 PORT LIST
     S1 STRUCT
          A1 STR (8)
          A2 LIST (4)
               B2 STR (6)
     END;
```

the following FOR-loop will not work:

```
FOR L1, L2
     FOR A2, A2
          S1=S1
     END
END;
```

The A2 lists are in use by the inner FOR-loop (FOR A2.B2,A2.B2) when the assignment S1=S1 is encountered. The datacomputer expands S1=S1 internally into:

```
A1=A1
FOR L1.S1.A2,L2.S1.A2
     B2=B2
END;
```

This constitutes a second use of the A2 lists, which cannot be handled.

2. In Version 0/11 of datalanguage, there is one general restriction on sequences of nested FOR-loops, which can be stated as follows:

Sequences of nested FOR-loops are restricted to be a number (possibly 0) of FOR-loops without output LISTs, followed by an arbitrary number, at least 1, of FOR-loops with output LISTs.

For example,

```
 FOR A                     FOR A                   FOR A
   FOR B,C                   FOR B
(ASSIGNMENT)
      (ASSIGNMENT)             FOR C,D              END;
   END;                          (ASSIGNMENT)
 END;                          END;
                             END;
                           END;
```

The first two examples are legal, whereas the third is not.

3. A FOR-loop with no output LIST can contain only one datalanguage statement as the FOR-body, not a series of statements. Because of restriction 2, that one statement must be a FOR.

This does not apply to a FOR with an output LIST.

4. The only comparison operators which can be evaluated from an inversion are EQ and NE. All other comparison operators must be evaluated by a linear search through a set of records. If the container being compared is a member of an inner list, only the EQ comparison operator can be evaluated from an inversion.

5. It is impossible to assign members of a LIST without setting up a FOR-loop (either explicitly or implicitly). For example, given the PORT is:

```
   CREATE L1 PORT LIST (5)
        S1 STR (3);
```

The following assignment is illegal:

```
   S1='FOO';
```

because it treats the five members of S1 as if they were a single data item.

6. Two outermost containers with the same name may not be open at the same time. This is true even though the containers may have different pathnames in the directory.

7. If an output PORT is punctuated, all assignments before each punctuation character must be completed before any assignments are made after the punctuation character. That is, the datacomputer cannot back up over punctuation in an output PORT. For example, given an output PORT of the form:

```
   PP PORT LIST
        S1 STRUCT
             A1 STR (3),P=EOR
             A2 STR (3),P=EOR
        END
```

assignments must be made in the same order as the STRs appear in the STRUCT.

```
   A1='FOO';
   A2='BAR';
```

will take effect correctly, but

```
   A2='BAR';
   A1='FOO';
```

will not.

Because of the internal paging of the datacomputer, PORTs with STRUCTs containing long STRs (i.e. greater than 2560 ASCII characters) have a similar restriction. for example, the LIST

```
FF PORT LIST
    S1 STRUCT
        A1 STR (10000)
        A2 STR (10000)
        A3 STR (10000)
    END
```

may have assignments done only in the same order as they appear in the STRUCT.

8. The datacomputer checks all descriptions at creation time to make sure that the byte parsing algorithm can be followed. Whenever a subcontainer byte size differs from the parental byte size the following tests are made (in order); if it is accepted by (any) test, the subcontainer is accepted:

1) If the entire subcontainer fits within the remainder of the parent byte, it is accepted. thus, in example 1 (below), the STRing "S" is accepted.

2) If a parental byte boundary could be crossed by the subcontainer, it must be aligned on its own byte boundary. In example 2, the description is rejected because "S" crosses the boundary defined by R2, and starts on an 18-bit rather than 7-bit boundary. In example 3, since "S" is aligned on a 7 bit boundary, it is accepted.

```
ex 1:        R STRUCT, B=36
             I INT, B=18
             S STR(2), B=7
             END /* R */

ex 2:        R2 STRUCT, B=36
             I INT ,B=18
             S STR(7), B=7
             END /* R2 */

ex 3:        R3 STRUCT, B=36
             I INT , B=18
             PAD BYTE, B=3
             S STR(7), B=7
             END /* R3 */
```

3) The byte size of a subcontainer must be greater than the remainder of 36 divided by the parental byte size. This is because data is packed in the datacomputer into 36 bit words and parent alignment is not followed.

```
R STRUCT, B=32
S1 INTEGER ,B=4
S2 INTEGER ,B=16
```

```
S3 BYTE, B=12
S4 BYTE, B=4
END /* R */
```

in this example, S3 would be legal because it fits within the 32 bit parent byte, but S4 is not accepted because it would fit in the remaining bits after the 32 in the 36 bit word buffer.

4) If the parental byte size is less than 18 bits, all subcontainer byte sizes must evenly divide the parent byte size. Again, this is because the 18 bit parent bytes are packed into 36 bit words.

Please contact CCA if you need help with complicated byte structures.

Appendix F: Differences between 0/10 and 0/11

The following is a list of user specifiable (i.e. syntactic) differences between 0/10 and 0/11 datalanguage.

Additions

Container Address Tables -- the CAT option for LISTs

Inversions and optimized indexing for LISTs with variable length members

Virtual Data

LIST member indexing -- the V=I option for BYTEs

Update (fixed length replacement) -- the UPDATE loop

WITH clauses on the left hand loop argument

Names on the right hand side of relational expressions

Integer Constants

Modifications

LIST (instead of LIST member) naming for loop arguments

Appendix G: Error Messages

The following is not a comprehensive listing of all the possible error messages the datacomputer can produce in response to bad datalanguage; and most users will only see a few of them. Many of the messages are self-explanatory. Some messages will contain, in an actual datacomputer session, a name specifing, for example, the unopened FILE/PORT which the user had assigned (CRER: RHS FILE/PORT NOT OPENED:) or the non-existant pathname to which the user had refered (CRER: LHS PATHNAME NOT FOUND:). The error messages are always one line; in an actual session they do not contain the carriage returns used in the following listing to increase legibility. The messages will also contain the date and time of the printout.

-U000	IOOPEN:	NET CONNECTION DIED
-U000	COCL:	BAD CLOSE ARGUMENT
-U000	COCL:	NODE SETS NOT ALLOWED
-U000	COCL:	FILE/PORT NOT OPEN
-U000	COCL:	END OF STATEMENT EXPECTED
-U000	COCN:	NODE SETS NOT ALLOWED
-U000	COCN:	CLOSED OR NOT A PORT
-U000	COCN:	BAD HOST/FILE SPECIFICATION
-U000	COCN:	BAD HOST-SOCKET SPECIFICATION
-U000	COCN:	SURROUND HOST NAME WITH SINGLE QUOTES
-U000	COCN:	ZERO SOCKET NOT ALLOWED
-U000	COCN:	END OF STATEMENT EXPECTED
-U000	COCN:	ZERO HOST-SOCKET NOT ALLOWED
-U000	COCO:	CONFLICTING GRANT AND DENY PRIVILEGES
-U000	COCO:	"," EXPECTED
-U000	COCO:	BAD PRIVILEGE TUPLE OPTION
-U000	COCO:	REDUNDANT USER ID
-U000	COCO:	BAD USER-ID
-U000	COCO:	REDUNDANT HOST
-U000	COCO:	BAD HOST NUMBER
-U000	COCO:	REDUNDANT SOCKET
-U000	COCO:	BAD SOCKET NUMBER
-U000	COCO:	REDUNDANT PASSWORD
-U000	COCO:	BAD PASSWORD
-U000	COCO:	REDUNDANT GRANT PRIVILEGES
-U000	COCO:	BAD GRANT PRIVILEGE
-U000	COCO:	REDUNDANT GRANT PRIVILEGE
-U000	COCO:	REDUNDANT DENY PRIVILEGES

-U000	COCO:	-C NOT ALLOWED
-U000	COCO:	BAD DENY PRIVILEGE
-U000	COCO:	REDUNDANT INDEX OPTION
-U000	COCO:	BAD INDEX
-U000	COCP:	NODE SETS NOT ALLOWED
-U000	CODE:	BAD DELETE OPTION
-U000	CODE:	$TOP NOT ALLOWED
-U000	CODE:	END OF STATEMENT EXPECTED
-U000	CODI:	NODE SETS NOT ALLOWED
-U000	CODI:	CLOSED OR NOT A PORT
-U000	CODI:	END OF STATEMENT EXPECTED
-U000	CODP:	NODE SETS NOT ALLOWED
-U000	CODP:	BAD PRIVILEGE TUPLE INDEX
-U000	CODP:	END OF STATEMENT EXPECTED
-U000	COLG:	NODE SETS NOT ALLOWED
-U000	COLG:	FILE/PORT LOGIN NOT ALLOWED
-U000	COLG:	END OF STATEMENT EXPECTED
-U000	COLI:	BAD LIST OPTION
-U000	COLO:	END OF STATEMENT EXPECTED
-U000	COLO:	$OPEN $PRIV NOT IMPLEMENTED
-U000	COLP:	BAD LIST OPTION
-U000	COLP:	NAME NOT FOUND
-U000	COLP:	END OF STATEMENT EXPECTED
-U000	COLP:	UNOPENED $DESC NOT IMPLEMENTED
-U000	COLP:	"*" $DESC NOT IMPLEMENTED
-U000	COLP:	"**" $DESC NOT IMPLEMENTED
-U000	COLP:	"*" $ALLOC NOT IMPLEMENTED
-U000	COLP:	"**" $ALLOC NOT IMPLEMENTED
-U000	COLP:	"*" $PRIV NOT IMPLEMENTED
-U000	COLP:	"**" $PRIV NOT IMPLEMENTED
-U000	COMD:	NODE SETS NOT ALLOWED
-U000	COMD:	FILE/PORT NOT OPEN
-U000	COMD:	BAD MODE OPTION
-U000	COMD:	END OF STATEMENT EXPECTED
-U000	COMD:	NO DEFERRED READ
-U000	COMD:	PORTS CANNOT BE DEFERRED
-U000	CONL:	NO PASSWORD FOR TOP NODE
-U000	CONL:	BAD PATHNAME
-U000	CONL:	NAME (IDENT) EXPECTED
-U000	CONL:	PASSWORDS IN OPEN PATHNAMES NOT ALLOWED
-U000	COOP:	NODE SETS NOT ALLOWED
-U000	COOP:	FILE/PORT ALREADY OPEN
-U000	COOP:	BAD MODE OPTION
-U000	COOP:	NO DEFERRED READ
-U000	COOP:	END OF STATEMENT EXPECTED
-U000	COOP:	CANNOT OPEN FILE/PORT
-U000	COPP:	BAD PASSWORD SPECIFICATION
-U000	CRCM25:	CAN'T FIND CONTEXT
-U000	CRER:	RHS FILE/PORT NOT OPEN:
-U000	CRER:	LHS FILE/PORT NOT OPEN:
-U000	CRER:	RHS PATHNAME NOT FOUND:
-U000	CRER:	LHS PATHNAME NOT FOUND:

-U000	CRTN:	OPERATOR NODE EXPECTED
-U000	CRTN5:	INVALID OPCODE
-U000	DDCD:	TEMPORARY CANNOT BE SUBNODE
-U000	DDCD:	BAD OUTER CONTAINER SPECIFICATION
-U000	DDCD:	BAD PATHNAME SPECIFICATION
-U000	DDCD:	NAME EXPECTED
-U000	DDCD55:	THERE IS AN OPEN FILE/PORT WITH SAME NAME
-U000	DDCD60:	DATA TYPE EXPECTED
-U000	DDCD:	INNER LISTS NEED DIMENSION
-U000	DDSI:	INNER LEVEL STRINGS NEED COUNT
-U000	DDCT:	NUMBER OR "," EXPECTED
-U000	DDCT:	NUMBER EXPECTED
-U000	DDCT:	MAX COUNT MUST BE LARGER THAN MIN
-U000	DDCT:	")" EXPECTED
-U000	DDKO:	BAD KEYWORD OPTION
-U000	DDKO:	REDUNDANT ALIGNMENT SPECIFICATION
-U000	DDKO:	BAD DESCRIPTOR FOR ALIGNMENT
-U000	DDKO:	BAD ALIGNMENT OPTION
-U000	DDKO:	REDUNDANT BYTE SIZE SPECIFICATION
-U000	DDKO:	BAD DESCRIPTOR FOR BYTE SIZE
-U000	DDKO:	BAD BYTE SIZE
-U000	DDKO30:	BAD DATATYPE FOR COUNT-IN-DATA
-U000	DDKO:	REDUNDANT VARIABILITY SPECIFICATION
-U000	DDKO:	BAD DESCRIPTOR FOR COUNT-IN-DATA
-U000	DDKO30:	BAD COUNT-IN-DATA SIZE
-U000	DDKO:	REDUNDANT VARIABILITY SPECIFICATION
-U000	DDKO:	BAD DATATYPE FOR DELIMITER
-U000	DDKO:	BAD DESCRIPTOR FOR DELIMITER
-U000	DDKO:	DELIMITER MUST BE STRING OR INTEGER
-U000	DDKO:	DELIMITER CAN ONLY BE ONE CHAR
-U000	DDKO:	REDUNDANT FILLER SPECIFICATION
-U000	DDKO:	BAD DESCRIPTOR FOR FILLER
-U000	DDKO:	FILLER CAN ONLY BE ONE CHAR
-U000	DDKO:	REDUNDANT INVERSION SPECIFICATION
-U000	DDKO:	NONINVERTIBLE CONTAINER
-U000	DDKO60:	NONINVERTIBLE CONTAINER
-U000	DDKO:	BAD DESCRIPTOR FOR INVERSION
-U000	DDKO:	BAD INVERSION OPTION
-U000	DDKO60:	NOT A LIST MEMBER
-U000	DDKO62:	INVERTED GRANDCHILDREN NOT READY
-U000	DDKO63:	I=D ALLOWED ONLY ON OUTER LEVEL LIST MEMBERS
-U000	DDKO:	LENGTH IN DATA NOT IMPLEMENTED
-U000	DDKO:	BAD DATATYPE FOR LENGTH IN DATA
-U000	DDKO:	REDUNDANT VARIABILITY SPECIFICATION
-U000	DDKO:	BAD DESCRIPTOR FOR LENGTH IN DATA
-U000	DDKO:	BAD LENGTH IN DATA SIZE
-U000	DDKO80:	REDUNDANT VARIABILITY SPECIFICATION
-U000	DDKO80:	BAD DESCRIPTOR FOR PUNCTUATION
-U000	DDKO80:	BAD PUNCTUATION OPTION
-U000	DDKO90:	BAD DATATYPE FOR VIRTUAL DATA
-U000	DDKO90:	VIRTUAL CONTAINERS ARE NOT

		INVERTIBLE
-U000	DDKO:	BAD DESCRIPTOR FOR VIRTUAL DATA
-U000	DDKO:	BAD VIRTUAL DATA OPTION
-U000	DDKO:	ONLY LISTS HAVE CATS
-U000	DDKO:	ONLY OUTER LEVEL LISTS HAVE CATS
-U000	DEID:	BAD $TOP SPEC
-U000	DEID:	END OF STATEMENT EXPECTED
-U000	DEOD:	ERROR IN REPARSING CD
-U000	DFDD:	TOP LEVEL COUNTED & DELIMITED THINGS DON'T WORK
-U000	DFDT:	VARIABILITY REQUIRES TERMINATOR
-U000	DFFC:	OUTER BYTE SIZE SMALLER THAN INNER
-U000	DFFC4:	ASCII DATA REQUIRES ASCII FILLER
-U000	DFFC06:	BYTESIZE TOO SMALL FOR FILLER
-U000	DFFC15:	BYTESIZE TOO SMALL FOR DELIMITER
-U000	DFFC20:	BYTESIZE TOO SMALL FOR SIZE IN DATA
-U000	DFFC60:	BAD VIRTUAL DATA OPTION
-U000	DFFC75:	BAD PUNCTUATION HIERARCHY
-U000	DFFF:	ONLY ONE INVERTED LIST, PLEASE
-U000	DFFF:	NON-ASCII PORT HAS NON-EOF
-U000	DFFF15:	PUNCTUATION ILLEGAL IN NON-ASCII PORT
-U000	DFFF38:	BAD PUNCTUATION HIERARCHY
-U000	DFFF38:	INFERIOR PUNCT BUT NO INFERIORS
-U000	DFFF42:	BAD PUNCTUATION HIERARCHY
-U000	DFFF50:	BAD PUNCTUATION HIERARCHY
-U000	DFLA40:	ONE MEMBER EXCEEDS DEFAULT SIZE
-U000	DIAN:	CANNOT OPEN FILE FOR INITIALIZATION
-U000	LAEX:	INTEGER CONSTANT OVERFLOW
-U000	LAEX:	CRLF NOT ALLOWED IN STRINGS
-U000	LPAS:	UNKNOWN COMMAND
-U000	LPAS:	NAME EXPECTED
-U000	LPAS:	END OF STATEMENT EXPECTED
-U000	LPAS:	CONSTANT EXPECTED
-U000	LPBP:	")" EXPECTED
-U000	LPFOR:	NAME EXPECTED
-U000	LPFOR:	BAD STATEMENT INSIDE A FOR LOOP
-U000	LPFOR:	NULL FOR-BODIES NOT PERMITTED
-U000	LPNN:	IDENTIFIER EXPECTED IN PATHNAME
-U000	LPRE:	PATHNAME EXPECTED
-U000	LPRE:	BAD RELATION
-U000	LPRE:	PATHNAME OR CONSTANT EXPECTED
-U000	LPSY:	ILLEGAL REQUEST
-U000	LPSY:	UNKNOWN REQUEST
-U000	LPSY:	END OF STATEMENT EXPECTED
-U000	SAAN:	IMPLIED LIST INAPPROPRIATE
-U000	SAAS:	FUNNY NODE TYPE
-U000	SAAS10:	NO MATCH - BAD TYPE FOR
-U000	SAAS10:	NO MATCH - NO MATCHING MEMBERS FOR
-U000	SAAS:	CAN'T USE LITERALS WITH BIG STRINGS
-U000	SAAS30:	FORARG MUST BE LIST
-U000	SAAS30:	NO MATCH--BAD TYPE FOR
-U000	SAAS30:	NO MATCH--NO MATCHING MEMBER FOR

-U000	SACR:	OPERATOR NODE EXPECTED
-U000	SACR:	DIFFERENT INNER LISTS REPRESENTED
-U000	SACR:	NESTED 'ANY'S NOT IMPLEMENTED
-U000	SAFR:	CAN'T HAVE 'FOR' INSIDE OF 'UPDATE'
-U000	SAFR:	CRUFTY FORARGS
-U000	SAFR05:	FORARG MUST BE LIST
-U000	SAFR25:	FORARG MUST BE LIST
-U000	SAFR42:	DATALANGUAGE TOO COMPLICATED
-U000	SAGM:	BAD DATATYPE
-U000	SAMA:	NULL SYNTAX TREE POINTER
-U000	SAMA:	NOT OPERATOR NODE
-U000	SAMA:	BAD SYNTAX TREE OP CODE
-U000	SARE:	NAME NODE EXPECTED
-U000	SARE:	LEFT SIDE NOT STRING
-U000	SARE:	NAME OR STRING NODE EXPECTED
-U000	SARE:	LIST LEVELS DON'T MATCH
-U000	SASR:	OPERATOR TYPE EXPECTED
-U000	SASR:	BAD GRAPH OP CODE
-U000	SAUP:	CAN'T HAVE 'UPDATE' IN 'FOR'
-U000	SAUP05:	FORARG MUST BE LIST
-U000	SAUP08:	MISMATCHED 'UPDATE'S LOSE
-U000	SAUP08:	EXPECTING TRANSACTION LIST
-U000	SAUP10:	FORARG MUST BE LIST
-U000	SAWI:	NOT WITH NODE
-U000	SAWI45:	NO UNINVERTED PART
-U000	GGGOF2:	UPDATE REQUIRES WRITE MODE
-U000	GGGOF:	OUTPUT MODE IS NOT WRITE OR APPEND
-U000	GGGOF:	NEITHER READ NOR WRITE
-U000	GGGOF:	BOTH READ AND WRITE SAME
-U000	GGUP:	ZERO LENGTH GRAPH NODE
-U000	SBAR82:	RAN OUT OF FILE BLOCKS
-U000	SBFR:	LIST ALREADY IN USE (OLD CGAR CASE)
-U000	SBFR:	INCOMPATIBLE LIST COUNTS
-U000	SBFR:	LIST ALREADY IN USE (OLD CGAR CASE)
-U000	SBFR:	USE FLAG GOT RESET
-U000	SBIB:	OPERATOR TYPE EXPECTED
-U000	SBIB:	INVERTED BIT NOT SET
-U000	SBIB:	BAD GRAPH OP CODE
-U000	SBIB:	UNIARY AND/OR
-U000	SBIB:	INDEX EXPECTED BUT NOT FOUND
-U000	SBIF:	OPERATOR TYPE EXPECTED
-U000	SBIF:	INVERTED BIT NOT SET
-U000	SBIF:	BAD GRAPH OP CODE
-U000	SBIF:	INDEX EXPECTED BUT GOT GIBBERISH
-U000	SBMA:	OPERATOR NODE EXPECTED
-U000	SBMA:	CRUFTY OPCODE
-U000	SBMA10:	CAN'T HAVE CONSTANT ON LEFT SIDE
-U000	SBNN:	NAME NODE EXPECTED
-U000	SBOP:	CAN'T OPEN CONSTANTS
-U000	SBPP:	CRUFTY OPCODE
-U000	SBPP60:	CONSTANT NOT ALLOWED
-U000	SBSR:	CAN'T FIND TOP NODE
-U000	SBUP:	LIST ALREADY IN USE

-U000	SBUP:	IN-USE FLAG GOT RESET
-U000	CHEB:	CAN'T START NON-EXISTENT CONTAINERS
-U000	CHEB50:	ILLEGAL REFERENCE TO LIST MEMBER
-U000	CHEB70:	ILLEGAL REFERENCE TO LIST MEMBER
-U000	CHEB80:	INDEX NOT IN LIST
-U000	CHEE:	ARGS NOT MATCHED
-U000	CHE[illegible]:	SKIP STUFF NEEDED ??
-U000	CHMB:	IN-USE BIT IS NOT SET
-U000	CHME:	IN-USE BIT NOT SET
-U000	GHAN:	PUNCTUATION IN CONDITIONAL MEMBER
-U000	GHAN:	LIST IN USE
-U000	GHANF:	BACK CHAIN EXPECTED
-U000	GHANF:	SOMEBODY ZORCHED THE IN-USE BIT
-U000	GHAS:	ILLEGAL ATTEMPT TO CHANGE VARIABLE LENGTH CONTAINER
-U000	GHAS25:	NO CAN DO; COME BACK NEXT YEAR (UPDATING INVERTED CONTAINERS THAT IS)
-U000	GHFB:	NO BACK LINKED LIST
-U000	GHFB:	IN-USE BIT NOT SET
-U000	GHFTI:	IN-USE BIT ALREADY SET
-U000	GHFTO:	IN-USE BIT ALREADY SET
-U000	GHIF:	OPERAND NOT A CONSTANT
-U000	GHNS:	PUNCTUATION BRANCHING GLITCH
-U000	GHPD:	BIG COMPARE NOT IMPLEMENTED
-U000	GHPT:	ZERO BRANCH ADDRESS
-U000	GHRUN:	BAD GRAPH NODE TYPE
-U000	GHUBM:	PAGE/PUNCTUATION BRANCHING PROBLEM
-U000	GHUEU:	IN-USE BIT NOT SET
-U000	GHUTT:	IN-USE BIT ALREADY SET
-U000	GTCN:	ILLEGAL CONVERT CODE
-U000	GTES:	BAD MODE
-U000	GTSB:	BAD CGRF/CGRR
-U000	GTSS:	BAD CGRF/CGRR
-U000	IGTU39:	MISSING TUPLE JUMP ADDRESS
-U000	IGTU49:	MISSING TUPLE JUMP ADDRESS
;U000	DFCB15:	BYTE BOUNDARY PROBLEM IN
;U000	DFCB20:	TAIL TOO LONG FOR
;U000	DFCB20:	BYTESIZE MUST DIVIDE PARENT FOR
;U000	DFCB40:	BYTE BOUNDARY GLITCH IN

Index